

Дистрибутив TurboDelphi (бесплатной версии Delphi), примеры и файлы проектов, готовые программы

САМОУЧИТЕЛЬ

Delphi

В. Рубанцев

в примерах, играх и программах

Дополнительные материалы

Цветные линии
Город
Молекулярный конструктор
Шариковая сборка

Логос
Логическая точечная головоломка

Японский кроссворд
Комбинаторная раскраска

Хитори
Японская хитовая головоломка

Бокалы
Головоломка-фокус

Жизнь
Клеточный автомат Конвея

Флип-Флоп
XorGame ShortGame
Игры-оборотни

Валерий Рубанцев

Delphi в примерах, играх и программах:

От простых приложений, решения задач и до программирования интеллектуальных игр

Дополнительные материалы

BEGIN

Баба с возу – кобыле легче!

Народная мудрость

Глава в корзину – книга легче!

Редакторская мудрость

В январе 2011 года в питерском издательстве *Наука и техника* вышла моя книга *Delphi в примерах, играх и программах*. К сожалению, в ней не нашлось места для нескольких глав, рассказывающих о программировании игр и головоломок.

Теперь я хочу восстановить социальную справедливость и вернуть приложения на их историческую родину. Мне кажется, они вполне того заслуживают.

Итак, здесь вы найдете подробнейшее, детальнейшее описание таких полнофункциональных игровых и головоломных приложений:

Японский кроссворд - популярнейшая головоломка на раскрашивание квадратиков.

Логос – замечательная логическая головоломка с точками.

Цветные линии, Город, Молекулярный конструктор – известная во всем мире игра на собирание шариков, а также ее естественнонаучные авторские вариации.

Жизнь – самый известный в мире клеточный автомат Джона Конвея. Играйте и размножайтесь!

Флип-Флоп, XorGame, ShortGame – превосходные игры-оборотни.

Бокалы – фокусная головоломка-перевертыш.

Хитори – одна из лучших современных японских головоломок, мировой хит!

Почти все проекты сопровождаются исчерпывающим *анализом* игр и головоломок. Поиск *выигрышных стратегий* и разработка *эффективных алгоритмов* - вот главное содержание и достоинство представленного здесь материала, поскольку в большинстве книг по программированию читателям сразу же предъявляется *готовое* решение проблемы, из которого невозможно (или, по крайней мере, весьма затруднительно) понять, откуда у него ноги растут и где собака зарыта.

С моей точки зрения, гораздо полезнее объяснить именно процесс *предварительных* исследований проблемы, чем описать реализацию полученного решения на каком-либо языке программирования, так как это исключительно дело техники и обычно не вызывает затруднений даже у начинающих программистов.

Несмотря на обилие материала, за пределами книги осталось еще очень много любопытных проблем, способных серьезно заинтересовать любителей программирования. В ближайшие месяцы я планирую на своем сайте *rvGames.de* выложить несколько книг по программированию. Заходите!

Валерий Рубанцев

Обозначения, принятые в книге:



Дополнение, примечание



Предложение, ненавязчивое требование



Предупреждение



Задание для *самостоятельного* решения



Папка с *исходным кодом* приложения

Исходный код почти всех проектов, описанных ниже, вы найдете на *DVD,* прилагаемом к «бумажной» книге. Остальные исходники ждут вас в папке *_SourceMin*.

А *полный* набор исходных кодов находится в папке *_Source*.

Официальный сайт книги: www.rvGames.de

Адрес для писем и газет: rvGames@mail.ru

Оглавление

Занятие 5. ДОСКОнально о сетке, или Занимательное полеводство

Игры-оборотни

В этом разделе книги речь пойдёт, разумеется, не о тех играх, в которых фигурируют вурдалаки, докторы хайды, вовкулаки, вервольфы и прочая нечисть. Всё гораздо милее и интереснее – фишки для таких игр окрашены в *два* цвета (например, с одной стороны они белые, с другой – чёрные) и время от времени они *изменяют* свой цвет на противоположный (иначе говоря, *переворачиваются* на другую сторону).

Классическим образцом такой игры является *реверси* (*отелло*), но эта игра заслуживает, конечно, гораздо большего внимания, чем мы можем уделить ей в данной книге. Как ни странно, все другие игры-оборотни значительно менее интересны и известны, поэтому мы займёмся программированием *головоломок*-оборотней. Зато мы убьём ещё парочку зайцев: закрепим на практике знания о компоненте *TDrawGrid* и продолжим совершенствовать мастерство изготовления кнопок из подручного материала.

XorGame, или Как обнулить матрицу

Сказать по правде, так и головоломок-оборотней придумано не очень много. А между тем, как говаривал гроссмейстер товарищ Бендер, вертеть фишки – *плодотворная дебютная идея*! Все известные головоломки отличаются друг от друга в основном правилами переворачивания фишек. Значит, достаточно написать программу для решения одной головоломки, а затем модифицировать её для всех остальных.

А начнём мы, пожалуй, с самого известного «оборотня», который называется *Флип-Флоп* (прояснить сущность этой игры нам поможет англо-русский словарь, в котором вы без труда найдёте и *flip*, и *flop*). Придумали её на фирме *Геймос* (*Gamos*), которая породила множество весьма полезных для ума игр. Эта же головоломка использована и в игре *Братья Пилоты. По следам полосатого слона*, где Шеф и Коллега должны открыть сейф, похожий на холодильник, поворачивая ручки, которые могут занимать вертикальное и горизонтальное положение.



В коллекции игр от *Home Games Center*, версия 10.19 от 2000 г. также имеется эта игра, но в более скромном обличии (Рис. 5.1).



Рис. 5.1. Не все простое гениально!

Здесь предлагается все крестики заменить галочками. Больше ничего нового нет.

Суть игры довольно проста: в некоем доме имеется 16 окон – по четыре на каждом из четырёх этажей. Некоторые окна *открыты*, остальные – *закрыты*. *Ваша цель* – открыть все окна настежь. Для этого вы можете поочерёдно открывать любые закрытые окна (или закрывать открытые). Но – по странной прихоти городского архитектора - все другие окна на том же этаже и в том же подъезде изменят своё состояние на противоположное (Рис. 5.2). Несмотря на «аскетизм» формулировки, задача не очень простая, в чём вы сами можете убедиться, похлопав створками окон.

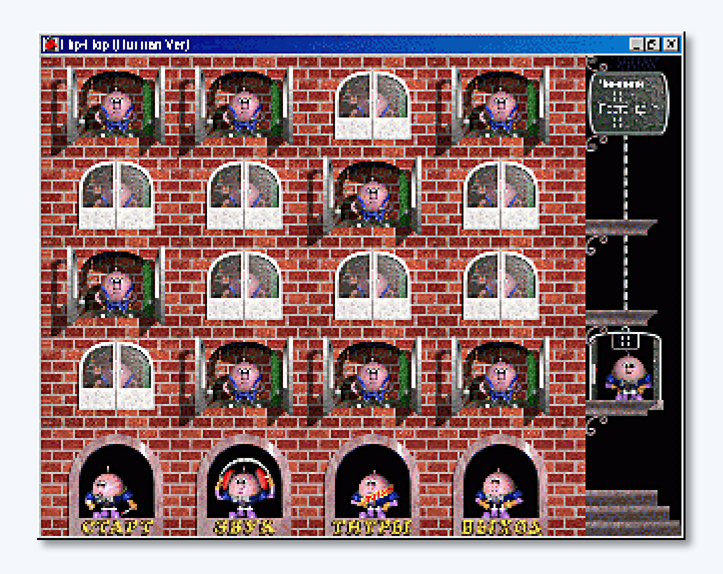


Рис. 5.2. Эх, флип-флопнем и окнами хлопнем!

В этой игре-головоломке нет никаких переворачивающихся фишек, но это ровным счётом ничего не меняет – для нас важно только то, что какие-то объекты могут находиться в двух «противоположных» состояниях. Например, окно может быть либо открыто, либо закрыто, фишка может быть либо чёрной, либо белой, и так далее. Совершенно естественно возникает желание обозначить одно состояние объекта *единицей*, а другое *нулём*, не различая при этом особенностей реализации этих состояний в конкретной программе. А игровое поле удобно описать в виде *двумерного массива*.

Обозначив закрытые окна *единицей*, а открытые – *нулём*, мы можем условие задачи переписать так (Рис. 5.3).

0	0	1	0
1	1	0	1
0	1	1	1
1	0	0	0

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Рис. 5.3. *Начальная* (исходная) позиция - *Конечная* позиция

Сформулируем правила игры в «матричном» виде.

На квадратном игровом поле размером 4 на 4 клетки расположены числа 1 и 0. Задача игрока заключается в том, чтобы за *наименьшее* число ходов перевести *начальную* позицию на игровом поле в *конечную*, состоящую из одних нулей. Чтобы сделать *ход*, нужно любое из чисел на поле заменить противоположным, то есть единицу – нулём, а нуль – единицей. При этом все остальные числа в том же столбце и в той же строке также изменяют своё значение на *противоположное*.

Предварительные изыскания

Все эти размышления мои
до сих пор предварительные были
не более как одною глупостью моею…

Н.Лесков, *Соборяне*

Можно предположить, что и эта и другие задачи в игре *Флип-Флоп* имеют решения. Очевиден ли для вас тот факт, что *любую* начальную позицию можно перевести в конечную? Если это так, то вы смело можете пропустить несколько следующих страниц, на которых мы попробуем убедиться в этом, написав небольшую программу.

Кстати говоря, приём, с помощью которого мы будем перечислять позиции, очень даже пригодится вам при работе над такими играми, как лабиринты, сокобан или *Цветные линии*.

Размеры игрового поля мы обозначим своими именами и сохраним в *константах*:

```
const
  //размеры поля:
  POLE_WIDTH=4;
  POLE_HEIGHT=4;
```

Опишем массив поля в глобальной переменной типа *Byte*, чтобы зря не расходовать память компьютера:

```
var
  //массив поля:
  masPole: array[0..POLE_WIDTH-1, 0..POLE_HEIGHT-1] of byte;
```

Всего нулей и единиц на поле 16 штук, что в точности равняется числу битов в одном двухбайтовом слове, которое имеет тип *Word*. Нам потребуется при расчётах и число, на единицу меньшее этого (попросту – 15). Заведём для него *константу*

```
  //число клеток на поле-1:
  AllCells= POLE_WIDTH*POLE_HEIGHT-1;
```

Легко также определить, что всего различных позиций на поле может быть 2^{16} = 65536. Нам остаётся только закодировать каждую позицию двухбайтовым словом. Для этого перепишем, например, заданную начальную позицию в одну строку: 0010110101111000. Так как каждое из

чисел теперь является битом, то все вместе они составляют двухбайтовое слово. Вы можете записать его и в 16-ричной форме - *$2D78*. Как это делается, мы уже рассматривали в игре *тетрис*. Аналогично кодируется и любая другая начальная позиция, а конечная позиция с одними нулями как раз и описывается нулём.

Мы договорились всякое клеточное поле представлять компонентом *TDrawGrid*, поэтому и поместим его на свежей форме, желательно в верхнем левом углу, чтобы он не портил мизансцену. С установкой свойств сетки (**Name** = dgPole) тоже никаких затруднений не предвидится:


```
ColCount = 4
RowCount = 4
DefaultDrawing = False
FixedCols = 0
FixedRows = 0
OnDrawCell = dgPoleDrawCell (процедуры будут описаны ниже)
OnMouseDown = dgPoleMouseDown
```

Остальные свойства вы можете установить по своему вкусу – они никак не влияют на работоспособность программы (только не забывайте о присущем вам здравом смысле).

Как же нам подсчитать число *разрешимых* позиций, если мы пока ещё не научились толком решать такие задачи? А нужно сделать ловкий ход – воспользоваться *ретроспективным* анализом, то есть рассматривать позиции, начиная с *заключительной* (конечной). К сожалению, объём книги не позволяет подробно остановиться на этом приёме, добавлю только, что его применение бывает очень полезно при решении шахматных и других подобных головоломок (задач).

Одно из самых важных понятий в ретроспективном анализе – *ранг позиции*. Это число ходов, которые нужно сделать из исходной (у нас это конечная) позиции в заданную. Легко сообразить, что нужная нам позиция с одними нулями имеет ранг *0*, так как она получается «сама собой», без всяких ходов. Присвоим также позициям, для которых ранг неизвестен, значение *–1*, чтобы отличать их от уже ранжированных.

Сделав ход из позиции с рангом 0, мы получим позицию с рангом 1. Так как этот ход может быть сделан в любую из 16 клеток, то всего существует 16 разных «предвыигрышных позиций» и столько же «первых» ходов. Давайте для начала научимся делать хотя бы один *ход* в нашей игре-головоломке.

При запуске программы мы инициализируем массив игрового поля, заполняя его нулями, то есть создаём конечную позицию с рангом 0:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
var i, j: integer;
begin
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      //очистить клетку:
      masPole[i,j]:= 0;
end;  //FormCreate
```


Как обычно, все изменения в массиве поля нужно отобразить на экране, закрасив каждую клетку поля своим цветом: та, что в массиве обозначена *нулём*, – пусть будет *белой*, *единицей* – *красной* (вы можете выбрать другие цвета):

```
//ОТРИСОВАТЬ ЯЧЕЙКУ СЕТКИ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger; Rect: TRect; State: TGridDrawState);
var
  i,j,n: integer;
begin
  //закрасить клетку своим цветом:
  case masPole[ACol, ARow] of
    0: dgPole.Canvas.Brush.Color:= clWhite;
    else dgPole.Canvas.Brush.Color:= clRed;
  end;
  dgPole.Canvas.FillRect(Rect);
end;  //dgPoleDrawCell
```

Чтобы сделать ход, нужно просто щёлкнуть (точнее – нажать кнопку мыши, иначе мы не сможем узнать её координаты) мышкой в любой клетке. С помощью метода сетки *MouseToCell* мы легко определим, в какую клетку поля был сделан ход. Обратите внимание, что столбец и строка для этой клетки возвращаются в переменных *ACol* и *ARow*, которые нужно объявить в процедуре, использующей этот метод. Чтобы иметь возможность вводить *любую* начальную позицию, а не только нулевую, мы нагрузим и *правую* кнопку мыши – нажимая её, мы просто изменяем цвет клетки на противоположный, не затрагивая других клеток поля. А настоящий ход выполняется при нажатии *левой* кнопки мыши. В этом случае мы меняем цвет всех клеток поля, которые лежат в том же столбце и строке, что и «щёлкнутая». Как это сделано, вы без труда сможете разобраться сами (в принципе, это можно сделать и иначе). Ну и наконец, мы обновляем поле на экране, чтобы отразить произошедшие на нём изменения:

```
//СДЕЛАТЬ ХОД
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button; Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  i, j: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  if ssRight in shift then //инвертируем цвет клетки:
    if masPole[ACol,ARow]= 1 then masPole[ACol,ARow]:= 0
    else masPole[ACol,ARow]:= 1
  else if ssLeft in shift then begin //инвертировать ряды
    for j:= 0 to POLE_HEIGHT-1 do
      for i:= 0 to POLE_WIDTH-1 do
        if (i=ACol) or (j=ARow) then
        //инвертировать клетку:
        masPole[i,j]:= abs(masPole[i,j]-1)
  end;
  dgPole.Invalidate;
end;  //dgPoleMouseDown
```


Запустите программу и удостоверьтесь, что она работает правильно. Вы даже можете сыграть в тот же *Флип-Флоп*, предварительно расставив правой кнопкой мыши красные клетки (они соответствуют, если вы помните, *закрытым* окнам). Настоятельно рекомендую вам попробовать решить задачу, приведённую в начале раздела, чтобы потом у вас было с чем сравнивать свои успехи.

Научившись делать ходы, мы возвращаемся к нашей задаче – подсчёту разрешимых позиций в этой головоломке. Теперь мы можем находить все 16 позиций с рангом 1. Взяв любую из них, мы получим позиции с рангом 2, сделав ход в одну из 15 клеток (конечно, нет никакого смысла ходить в ту же клетку, что и перед этим, иначе мы просто вернёмся на 1 ход назад). И вот здесь нас ждут первые неприятности: позиции начнут *повторяться*. Это следствие того, что результирующая позиция не зависит от порядка ходов. Вы легко можете убедиться в этом, если сходите, например, сначала в верхний левый угол, а затем в правый нижний, после чего вернётесь к чистой доске и выполните ходы в обратном порядке. Если мы будем считать *все* получающиеся позиции, то некоторые посчитаем несколько раз, а в таком подсчёте нет никакого смысла. Конечно, можно записывать все ходы и позиции для сравнения, чтобы исключить повторы, но позиций-то многие тысячи, так что нас ждёт удовольствие не из приятных! Будет правильнее поручить ведение протокола компьютеру, он с этим делом легко справится. Но прежде мы должны научить его переходить от одной позиции к другой и записывать нужную нам информацию.

Для анализа головоломки нам нужно запомнить ранг каждой позиции, координаты клетки, в которую был сделан ход, приведший к ней, и предыдущую позицию, чтобы мы могли проследить всю цепочку ходов. Для того чтобы все данные хранились в *одном* месте, мы заведём глобальную переменную *Status,* представляющую собой массив данных о каждой возникающей на поле позиции:

```
Status: array[word] of TStatus; //информация о позиции на поле:
type TStatus= Record
  Range: integer; //ранг позиции - ход, на котором
                  //впервые могла возникнуть данная позиция
  Hod: TPoint;    //координаты клетки поля, на которую сделан ход
  Pred: word;     //предыдущая позиция
end;
```


Алгоритм подсчёта позиций можно представить себе так. Мы знаем, что ранг всех позиций в начале равен –1, так как ни одна из них ещё не возникла на поле (кроме последней, с одними нулями – она получается сразу же, на «нулевом» ходу, и ранг ее равен 0). Из нулевой позиции мы можем сделать 16 ходов и получить все позиции с рангом 1. Естественно, позиция с рангом 2 может возникнуть только в результате хода из позиции с рангом 1, поэтому мы перебираем все возможные позиции и, отыскав ещё не посчитанную (её ранг равен -1), делаем ходы во все клетки поля. Если в результате этого возникает новая позиция, то мы присваиваем ей ранг 2 и записываем всю информацию о ней.

Точно так же мы делаем и следующие ходы (переходим к позициям очередного ранга) – до тех пор, пока на очередном ходе будет найдена хотя бы одна новая позиция.

Хранить число найденных позиций мы будем в переменной

```
  //всего найдено позиций:
  AllPos: cardinal;
```

Так как теперь все ходы за нас будет делать компьютер, то мы должны научить его этому. Результат хода зависит от расстановки «фишек» на поле и от координат клетки, в которую производится ход. Возникающую после хода позицию нам нужно вернуть в процедуру, подсчитывающую позиции, поэтому все премудрости выполнения хода следует поместить в *функцию*. «Ручной» ход мало чем отличается от машинного, единственное отличие – нужно «упаковать» единицы и нули на поле в двухбайтовое слово, которое однозначно описывает позицию.

Делается это так. Находим в двухбайтовом слове *w*, описывающем текущую позицию, те биты, которые соответствуют переворачиваемым столбцу и строке, и инвертируем их с помощью логической операции *xor* (это «магическое» слово и было использовано для именования головоломки). Чтобы подобраться к этой позиции в слове *w*, нужно сдвинуть единицу *влево* в нужную позицию логическим оператором *shl*. Например, если нужно инвертировать третью клетку поля, то следует передвинуть единицу в 13-ю позицию (нумерация битов в двухбайтовом слове такая: 15-14-13-12-11-10-9-8-7-6-5-4-3-2-1-0). В результате мы получим двоичное число 10000000000000, а операция *w xor 10000000000000* как раз и инвертирует нужный бит. Во всех остальных тонкостях этой функции разберитесь сами:


```
//ВЫПОЛНИТЬ ХОД
function XorBits(x, y: byte; w: word): word;
//x, y - координаты хода
//w - биты поля
var
  i, j: integer;
begin
  Result:= w;
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      if (i=x) or (j=y) then begin
        //инвертировать клетку:
        masPole[i,j]:= abs(masPole[i,j]-1);
        Result:= Result xor (1 shl (AllCells-POLE_WIDTH*j-i));
      end;
end;  //XorBits
```

Для прояснения действия этой функции, мы проведём её проверку на тех ходах, результаты которых легко проверить. Например, мы наизусть знаем все первые ходы из нулевой позиции. Давайте протестируем новую функцию на них. Установим на форме *кнопку sbtTesting*, нажав на которую мы сможем последовательно выполнить все 16 ходов.

Нелишним будет научиться по двухбайтовому слову *w*, описывающему поле, выводить «фишки» на экран . Для этого, начиная со *старшего* бита (соответственно, с самой *первой* клетки поля), мы определяем значение каждого бита слова и заносим его в массив поля. Закончив формирование массива, мы обновляем поле на экране. Обратите внимание на то, как выделяется старший бит слова *w*.

```
//ПОКАЗАТЬ ПОЛЕ
procedure TForm1.ShowPole(w: word);
//w - слово, описывающее позицию
var
```

```
  i, j: integer;
begin
  //занести в массив 1 или 0:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do begin
      if w and $8000 <> 0 then //- клетка фигуры
        masPole[i,j]:= 1
      else
        masPole[i,j]:= 0;     //- клетка "фона"
      w:=  w shl 1;
    end;
  //отрисовать сетку:
  dgPole.Invalidate;
end; //ShowPole
```


Процедура, обрабатывающая нажатие кнопки *sbtTesting*:

```
//ПРОВЕРКА
procedure TForm1.sbtTestingClick(Sender: TObject);
var
  i, j: integer;
  w: word;
begin
  //делаем ходы:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do begin
      w:= XorBits(i,j,0);
      ShowPole(w);
      showmessage (inttostr(w));
    end;  //for i
end; //sbtTestingClick
```

Запустите программу и удостоверьтесь, что, по крайней мере, из нулевой позиции ходы выполняются верно. Вы сможете продлить себе удовольствие, если вместо нулевой позиции возьмёте любую другую и проверите её. Всё это рано или поздно убедит вас, что функция работает правильно в *любой* позиции.

Теперь у нас есть всё необходимое для подсчёта позиций. Установите на форме *кнопку sbtCreateList* и напишите в обработчике её нажатия код:

```
//СОЗДАТЬ СПИСОК ПОЗИЦИЙ В МАССИВЕ
procedure TForm1.sbtCreateListClick(Sender: TObject);
var
  Hod: integer;
  flg: boolean;
  i,j,k,w: word;
begin
  Hod:= 0;
  //если ранг позиции равен -1, то она ещё не возникала на поле:
```

```
  for i:= 0 to 65535    do
    Status[i].range:= -1;

  //начальная позиция - её ранг равен 0:
  Status[0].range:= 0;

  //инициализируем счётчик найденных позиций:
  AllPos:= 1;

  //делаем следующие ходы:
  repeat
    flg:= False;
    inc(hod);
    listbox1.Items.add(inttostr(hod));
    for k:= 0 to 65535 do //- по всем возможным позициям
      if status[k].range= Hod- 1 then //- нашли позицию, возникшую
на
      begin                            //предыдущем ходу
      //делаем ходы из этой позиции:
      for j:= 0 to POLE_HEIGHT-1 do
        for i:= 0 to POLE_WIDTH-1 do begin
          w:= XorBits(i,j,k);
          if status[w].range= -1 then begin //- новая позиция
            //на этом ходу была найдена по крайней мере 1 позиция:
            flg:= True;
            //записать позицию в массив:
            Status[w].range:= hod;
            Status[w].hod.x:= i;
            Status[w].hod.y:= j;
            Status[w].Pred:= k;
            inc(AllPos);
          end;
        end;
      end;
  until flg= false; //продолжать, пока найдена хотя бы одна
                    //позиция на очередном ходу
```



```
  Showmessage('hodov= '+ inttostr(hod));
  Showmessage('AllPos= '+ inttostr(AllPos));
end;  //sbtCreateListClick
```

После всего содеянного у вас не должно возникнуть вопросов о том, *как* она работает. Поэтому запускайте программу и нажимайте кнопку. Очень скоро вы получите ответ на мучивший нас вопрос: все 65535 позиций (65536-я – это нулевая) «правильные» и для решения любой задачи потребуется не более 16 ходов (не удивляйтесь, что последний ход – 17-й, ведь он не дал ни одной позиции). Способ доказательства, конечно, «топорный», но зато и само доказательство железное!
Итак, мы «напрасно» потратили время – не найдено ни одной позиции, из которой нельзя было бы перейти к нулевой. Правда, мы теперь точно

знаем, что на решение задач может потребоваться от 1 до 16 ходов. А это уже кое-что: если вы сделали 16 ходов, а задачу так и не решили, значит, вы сделали *лишние* ходы.

Вот ещё бы узнать, *какие* ходы нужно сделать, чтобы решить задачу за минимальное число ходов!

А ведь у нас уже есть вся необходимая информация для этого! Она хранится в массиве *Status*, вот только пользоваться ей не очень удобно. Поэтому запишем все данные на диск. Для этого установите на форме *компонент TSaveDialog* TSaveDialog, ещё одну *кнопку – sbtSaveList* и наберите код для неё:


```
//ЗАПИСАТЬ ДАННЫЕ НА ДИСК
procedure TForm1.sbtSaveListClick(Sender: TObject);
var
  F: textfile;
  fn,s: string;
  i,j: integer;
  w: word;
begin
  //открыть файл для записи:
  savedialog1.DefaultExt:='txt';
  //расширение файла:
  savedialog1.Filter:='Данные (*.txt)|*.TXT';
  savedialog1.FilterIndex:=1;
  //записываем файл в папку res:
  s:=extractfilepath(application.exename)+'res\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите данные на диск';
  //имя файла по умолчанию:
  savedialog1.filename:='temp.txt';
  if not savedialog1.Execute then exit;
  //имя конечного файла:
  fn:= savedialog1.filename;
  assignfile(f,fn);
  rewrite(f);

  //записать данные:
  for i:=0 to 65535 do begin
    s:=''; w:= i;
    for j:= 0 to 15 do  begin
      //ставим пробел между четвёрками чисел:
      if j mod 4 = 0 then s:= s+' ';
      if w and $8000 <> 0 then //- клетка фигуры
        s:= s+'1'
      else
        s:= s+'0';              //- клетка поля
      w:=  w shl 1;
```

```
    end;
    s:=inttostr(i)+ '/'+s+'. Range= '+ inttostr(status[i].range)+'
Ход= '+         inttostr(status[i].hod.x)+'
'+inttostr(status[i].hod.y)+ ' Pred= '+inttostr(status[i].pred);
    if status[i].range<> -1 then writeln (f, s);
  end;

  //закрыть файл:
  closefile(f);
  messagebeep(0)
end; //sbtSaveListClick
```


Запустите программу, нажмите кнопку *Создать список*, а затем *Записать*. Если вы не изменяли значений по умолчанию, то сможете найти файл в *папке res* под именем *temp.txt*. Его начало выглядит так:

```
0/ 0000 0000 0000 0000. Range= 0 Ход= 0 0 Pred= 0
1/ 0000 0000 0000 0001. Range= 7 Ход= 3 3 Pred= 4382
2/ 0000 0000 0000 0010. Range= 7 Ход= 3 3 Pred= 4381
```

Загрузите файл в любой текстовый редактор и найдите помощью команды *Find* строку *0010 1101 0111 1000*, которая представляет собой *начальную* позицию игры *Флип-Флоп*. Вот что вы должны увидеть:

```
11640/ 0010 1101 0111 1000. Range= 8 Ход= 2 1 Pred= 602
```

Первое число – это наша позиция, записанная в 10-тичном виде. Четыре 4-значных числа после него – запись позиции по строкам. Затем идёт ранг позиции, ход (строка-колонка, отсчёт начинает с *нуля*) и, наконец, предыдущая позиция, записанная в 10-тичном виде.

Следовательно, в исходной позиции следует выполнить ход в клетку, находящуюся на пересечении второй колонки (так как мы считаем с нуля, то физически она третья слева; если вы путаетесь в координатах, то измените вывод информации в процедуре *sbtSaveListClick* так, чтобы самая левая колонка имела номер 1) и первой строки (второй сверху). В результате хода получится позиция, имеющая номер *602* в списке. Найдите её аналогично первой строке:

```
602/ 0000 0010 0101 1010. Range= 7 Ход= 3 3 Pred= 4933
```

Представим всю информацию из файла в удобном виде (Рис. 5.4).

```
11640/ 0010 1101 0111 1000. Range= 8 Ход1= 2 1 → 602
```

0	0	1	0
1	1	0	1
0	1	1	1
1	0	0	0

602/ 0000 0010 0101 1010. Range= 7 Ход2= 3 3 → 4933

0	0	0	0
0	0	1	0
0	1	0	1
1	0	1	0

4933/ 0001 0011 0100 0101. Range= 6 Ход3= 2 3 → 12650

0	0	0	1
0	0	1	1
0	1	0	0
0	1	0	1

12650/ 0011 0001 0110 1010. Range= 5 Ход4= 1 3 → 29989

0	0	1	1
0	0	0	1
0	1	1	0
1	0	1	0

29989/ 0111 0101 0010 0101. Range= 4 Ход5= 1 0 → 33121

0	1	1	1
0	1	0	1
0	0	1	0
0	1	0	1

33121/ 1000 0001 0110 0001. Range= 3 Ход6= 0 2 → 2457

1	0	0	0
0	0	0	1
0	1	1	0
0	0	0	1

2457/ 0000 1001 1001 1001. Range= 2 Ход7= 0 0 → 61713

0	0	0	0
1	0	0	1
1	0	0	0
1	0	0	1

61713/ 1111 0001 0001 0001. Range= 1 Ход8= 3 0 → 0

1	1	1	1
0	0	0	1
0	0	0	1
0	0	0	1

0   Range= 0

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Рис. 5.4. Делай, как я!

Теперь мы с уверенностью можем утверждать, что для решения предложенной задачи достаточно *восьми* ходов. И мы даже знаем, *какие* ходы следует делать, но до сих пор не выяснили, *почему* нужно ходить так, а не иначе. Попытаемся справиться и с этой проблемой.

Если вы подобным же образом проследите решение ещё нескольких задач, то наверняка обратите внимание на то, что ни один ход *не повторяется*. Когда это знаешь, легко найти объяснение данному «явлению»: повторный ход в ту же клетку просто «нейтрализует» первый ход, поэтому, выбросив любую пару одинаковых ходов, мы нисколько не изменим позицию, возникающую в результате выполнения остальных ходов. Отсюда слкдует вывод, что *любая задача может быть решена не более чем за 16 ходов*, в противном случае ходы неизбежно начнут *повторяться*. Ранее мы пришли к такому же выводу, произведя полный перебор позиций. Также нами было установлено, что порядок выполнения ходов не имеет никакого значения, поэтому существует *множество* решений нашей задачи, отличающихся только порядком ходов.

Вот ещё бы научиться определять те клетки, в которые нужно делать ходы! Это, пожалуй, самое трудное в этой головоломке, здесь вам потребуется уже вся ваша наблюдательность и сообразительность!

Последуем нашему принципу: не знаешь, с чего начать, начинай с конца! Вернёмся к *заключительной* позиции, состоящей из одних нулей. Какую бы клетку мы ни взяли, сумма чисел в соответствующей колонке и строке равна 0. Сделаем ход в клетку 3,0 (или в любую другую, безразлично). Получится такая позиция (Рис. 5.5).

1	1	1	1
0	0	0	1
0	0	0	1
0	0	0	1

Рис. 5.5. Первый ход

Во всех клетках с нулями сумма чисел станет равной 2. Во всех остальных, кроме угловой, 4. И только в угловой клетке сумма чисел будет равна 7. Запомните, что именно в неё мы и сделали ход.

Предположим, что мы сделали *второй* ход в одну из клеток с *нулями*. Сумма чисел в них равна двум. Значит, в соответствующей колонке и столбце 7 – 2 = 5 нулей (не посчитайте клетку на их пересечении дважды!). В результате хода появятся, наоборот, 5 единиц и 2 нуля, то есть сумма увеличится до пяти.

Если вы сделаете второй ход в клетку с *единицей* (исключая угловую, так как в неё второй раз ходить нельзя), то вместо четырёх единиц и трёх нулей возникнут 4 нуля и 3 единицы. Итак, в какую бы клетку мы ни сходили, сумма чисел в колонке и в строке, проходящих через неё, будет *нечётной*. Убедитесь самостоятельно, что чётность всех остальных клеток поля не изменяется: нечётные клетки (с нечётной суммой) так и останутся нечётными, а чётные (с чётной или нулевой суммой) также останутся чётными.

Для клеток, которые не лежат в колонке и столбце с «сыгравшей» клеткой, всё объясняется просто: в результате хода изменяются 2 числа – либо 0 + 0 → 1 + 1, либо 1 + 1 → 0 + 0, либо 1 + 0 → 0 + 1, поэтому сумма в чётных клетках изменится на чётное число (или останется прежней) и их чётность не изменится. Для нечётных клеток ситуация аналогичная.

В клетках, лежащих в упомянутых колонке и строке, изменится значение 4-х чисел. Если в колонке или строке было *чётное* количество единиц (или вообще не было), то количество нулей также было *чётным*. После выполнения хода чётность ряда не изменится. Рассуждая аналогично, можно доказать, что нечётный ряд также сохранит свою нечётность. Из этого следует, что в результате хода меняется чётность только *единственной* клетки – той, в которую сделан ход.

Сделав *третий* ход, мы изменим чётность соответствующей клетки. А так как ходить следует исключительно в чётные клетки, то с каждым ходом на поле будет добавляться одна нечётная клетка.

Задача из игры *Флип-Флоп* решается за 8 ходов. Вы можете легко проверить, что в исходной позиции ровно 8 нечётных клеток. Если мы теперь пойдем не от конечной позиции к начальной, а наоборот, то и ходить нам нужно в *нечётные* клетки. Тогда с каждым ходом их количество будет уменьшаться, и мы неминуемо придём к конечной позиции с одними нулями. Вот, как говорится, и весь сказ!

Благодаря проведённому анализу головоломки мы можем для *любой* начальной позиции указать число ходов, необходимых для её решения, и клетки, в которые эти ходы следует выполнять. В награду за умственное напряжение мы можем сделать себе послабление в виде дополнительного кода (выделенные строки) в процедуре вывода клеток игрового поля на экран, которая сама будет отмечать *плюсиком* нечётные клетки, и нам останется только усердно нажимать на них.

```
//ОТРИСОВАТЬ ЯЧЕЙКУ СЕТКИ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger; Rect: TRect; State: TGridDrawState);
var
  i,j,n: integer;
begin
  //закрасить клетку своим цветом:
  case masPole[ACol, ARow] of
    0: dgPole.Canvas.Brush.Color:= clWhite;
    else dgPole.Canvas.Brush.Color:= clRed;
  end;
  dgPole.Canvas.FillRect(Rect);

  //нечётный ряд?
  n:= 0;
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      if (i=ACol) or (j=ARow) then begin
        n:=n+masPole[i,j];
      end;
  //нечётный - вывести плюсик:
  if odd(n) then
  with Rect, dgPole.Canvas do
    textrect(Rect, left+(right-left-textwidth('+')) div 2,
top+(bottom-top-textheight('+')) div 2, '+');
end;  //dgPoleDrawCell
```

Запустите программу и правой кнопкой мыши введите исходную позицию. Картина, которую вы увидите, бесспорно, порадует настоящего ценителя головоломок (Рис. 5.6).

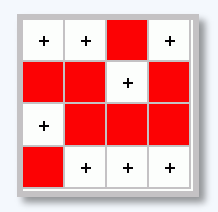


Рис. 5.6. Начальная позиция

Обидно только за игру *Флип-Флоп* - она пала жертвой нашего любопытства, в ней больше нет тайн для пытливого ума. А ведь хорошая игра, жаль с ней так быстро расставаться, поэтому давайте продлим очарованье и вдохнём в неё новую жизнь, усложнив последнюю себе тем, что чуточку подправим условия игры – в качестве конечной позиции возьмём не «завсегдашнюю» нулевую, а *произвольную*. Вы уже догадались, как нужно играть в этом случае?

Поскольку мы умеем любую позицию сводить к нулевой, то можно поступить так: от начальной позиции перейти к нулевой и от конечной также перейти к нулевой. Объединив все ходы, мы получим решение задачи в целом. Понятно, что решение некоторых задач при такой стратегии будет не самым коротким: если одна из позиций требует для «обнуления», например, 9 ходов, а вторая 10, то в сумме придётся затратить 19 ходов, что больше 16. Стало быть, мы сумеем из любой начальной позиции перейти в любую конечную, но при этом сделаем *лишние* ходы. Как же от них избавиться? Самые умные читатели могут отдохнуть, а мы пока составим процедуру для перехода от одной позиции к другой.

Установите на форме *кнопку sbtPerehod*, а в процедуре-обработчике напишите код, он очень простой, так как весь поиск кратчайшего перехода будет осуществляться в отдельной процедуре *Solution*. При её вызове нужно указать начальную и конечную позиции (в 10-тичном или 16-ричном виде), а также минимальное число ходов (если вы его не знаете, просто подбирайте, пока не найдёте самого короткого решения).

```
procedure TForm1.sbtPerehodClick(Sender: TObject);
begin
  //Solution(0, $e997, 2);
  //Solution(0, $1886, 3);
  //Solution(63897, 39409, 11);
  //Solution(0, 19, 5);
```

```
  //Solution(63897, 112, 5);
  //Solution(112, 39409, 6);
  //Solution($ffff, 0, 16);
  Solution(11640, 0, 16);
end;  //sbtPerehodClick
```

Процедуру поиска решения мы подробно рассматривать не будем – если она вас интересует, то постарайтесь разобраться в ней самостоятельно. Проследите решение нескольких задач и найдите причину, по которой предложенный нами ранее приём давал лишние ходы. Если вам не удастся справиться с этой проблемой, то читайте следующий раздел этой главы, в ней туман над *Флип-Флопом* рассеется окончательно.


```
//НАЙТИ ПЕРЕХОД ОТ ОДНОЙ ПОЗИЦИИ К ДРУГОЙ
procedure TForm1.Solution(w0, w1: word; nStep: integer);
//w0 - начальная позиция,
//w1 - конечная позиция,
//nStep2 - количество шагов (ходов)
label
  nextHod, nextCell;
var
  Hod: integer;
  nCells: integer;
  nCell: array[0..99] of integer;
  Pos: array[0..99] of word;
  Move: array[0..99] of integer;
  x, y: byte;
  w: word;

  procedure SavePos;
  var
    w: word;
    i, j: integer;
    x, y: integer;
    s: string;
  begin
    for i:= 1 to Hod do begin
      s:=''; w:=Pos[i];
      for j:= 0 to 15 do  begin
        if  w and $8000 <> 0 then //- клетка фигуры
          s:= s+'1'
        else
          s:= s+'0';                //- клетка "фона"
        w:= w shl 1;
      end;
     listbox1.items.add('Hod= '+ inttostr(i)+ ' Pos= '+ s);
     x:= Move[i] mod 4; y:= Move[i] div 4;
     listbox1.items.add('Move= '+ inttostr(x)+ ' ' +inttostr(y));
     listbox1.items.add('');
    end;
  end;
```

```
begin
  nCells:= 16;
  Hod:= 0; Pos[0]:= w0; nCell[0]:= -1; Move[0]:= -1;
nextHod:
  Inc(Hod); nCell[Hod]:= nCell[Hod-1];  Pos[Hod]:= Pos[Hod-1];
nextCell:
  inc(nCell[Hod]);
  if nCell[Hod]> nCells then begin //- прошли всё поле
    dec(Hod);
    if Hod< 0 then begin
      ShowMessage('OK');
      exit;
    end;
    Goto nextCell
  end;
  x:= nCell[Hod] mod 4; y:= nCell[Hod] div 4;
  w:= XorBits(x, y, Pos[Hod-1]);
  Pos[Hod]:= w;
  //запомнить ход:
  Move[Hod]:= nCell[Hod];

  if w = w1 then begin //- нашли!
    savepos;
    ShowPole(w);
    exit;
  end;

  if Hod< nStep then goto NextHod;

  goto nextCell
end;  //Solution
```


Напоследок – форма со всеми принадлежностями для успешной работы (Рис. 5.7).

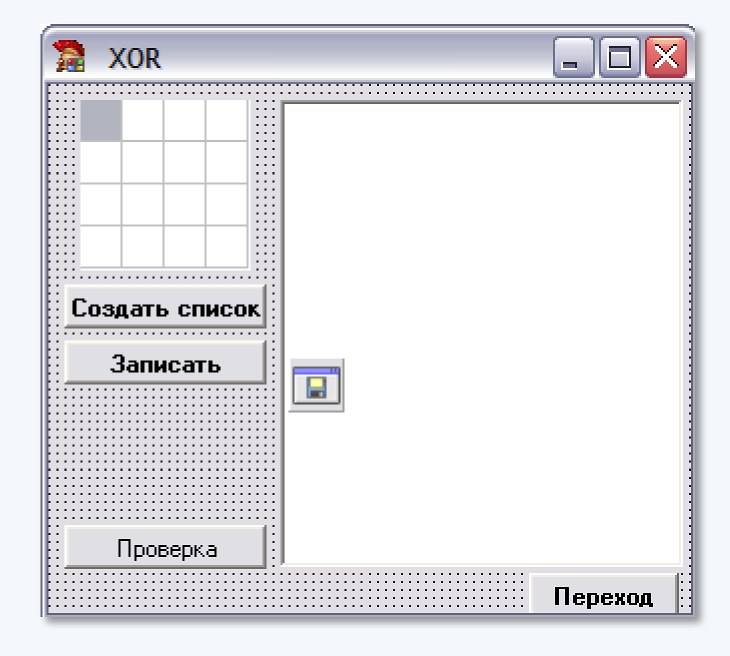


Рис. 5.7. Немудреный интерфейс!



Некоторые читатели могут упрекнуть автора в бесполезной трате времени и бумаги, ведь можно было сразу же перейти к программированию головоломки, а не демонстрировать «муки творчества». Действительно, обычно в книгах приводятся готовые, отлаженные программы, которые ясно свидетельствуют о недюжинных умственных способностях их авторов и создают у неразумных «чайников» превратное впечатление, что есть этакие «профессионалы», которые пишут программы в один присест, без напряжения и ошибок, и «любители», мучающиеся над каждой строкой исходного кода.

На самом-то деле умные мысли посещают и умных людей отнюдь не сразу (иначе компьютеры придумали бы ещё во времена Архимеда) и невозможны без предварительных «глупостей» и ошибок. Так как для вас, вероятно, гораздо важнее научиться писать *собственные* программы, чем переписывать чужие, то я и рискнул показать, какими окольными путями приходится подбираться к хорошим решениям и алгоритмам. Считайте, что вся проделанная нами работа – это черновик, который дальше мы будем использовать для создания «настоящей программы». (В скобках следует заметить, что и «черновик» тоже изрядно «отредактирован»: из него удалены очевидные заблуждения, опечатки и тупиковые мысли, так что не удивляйтесь, если ваши предварительные изыскания отнимут у вас гораздо больше времени и сил.)

Естественно, невозможно все программы разбирать так подробно, для этой мы сделали исключение - для примера, поэтому всегда имейте в виду, что прежде чем браться за написание игры, нужно предварительно отладить наиболее важные места вашей будущей программы.

Игра как она есть

Если для игры *Флип-Флоп* достаточно *одного* игрового поля, так как конечная позиция не изменяется и очень проста, то в *XorGame* ситуация осложняется тем, что конечная позиция в каждой партии *другая*. Естественно, держать в голове конечную позицию, когда нужно думать о выборе ходов, весьма затруднительно, поэтому для нашей игры мы сделаем *два* поля – с *начальной* позицией (размером побольше) и с *конечной* позицией (размером поменьше; вы можете сделать оба поля и равной величины, но в этом случае возможна неразбериха).

Поскольку с игрой *Флип-Флоп* мы досконально разобрались, то будет довольно скучно писать программу, которой нам пользоваться будет не интересно. Поэтому мы предусмотрим в нашей игре возможность *изменять* размеры поля – или вы считаете, что справитесь с головоломкой на любом поле? Достаточно иметь поля таких размеров: 4 x 4 (для начинающих), 5 x 5, 6 x 6, 7 x 7 (для головастых). Вы можете сделать поля ещё масштабнее, но ничего нового в игре от этого не появится.

Итак, откроем новый проект *XorGame* и займёмся *формой*:


```
Width = 397
Heightt = 312
BorderIcons = [biSystemMenu, biMinimize]
Caption = '  XorGame'
Icon – нарисуйте подходящий значок
PopupMenu = PopupMenu1 – добавим дальше
Position = poDesktopCenter
OnClose = FormClose – процедуры-обработчики.
OnCreate = FormCreate
```

Затем, оставив сверху место для кнопок, расположим на форме две сетки – *dgPole* и *dgPole2*. В целом форма со всеми компонентами должна выглядеть так (Рис. 5.8).

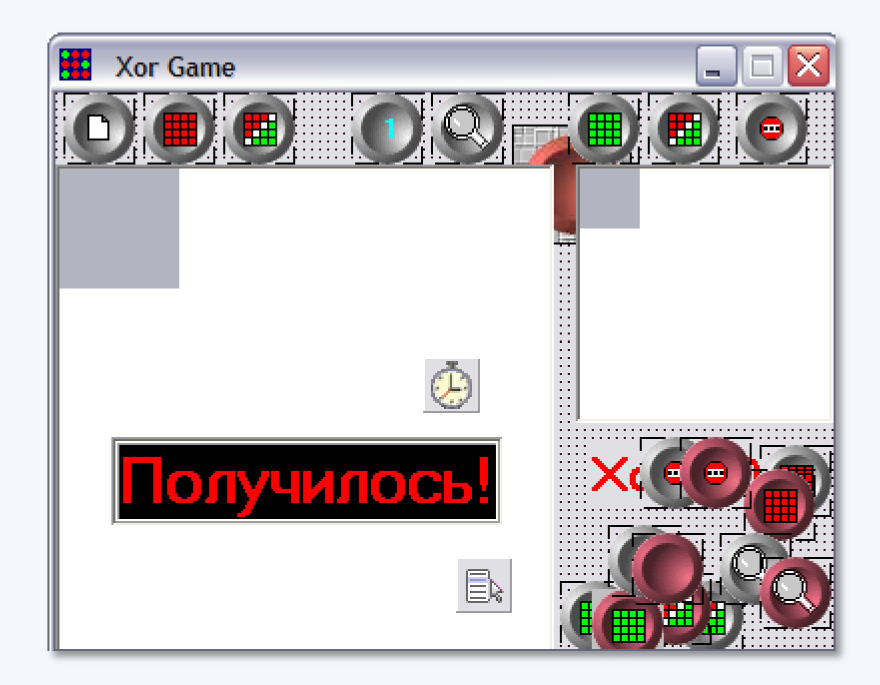


Рис. 5.8. Двупольная форма!

Свойства сетки *dgPole*:

```
Left = 0
Top = 36
Width = 249
```

```
    Height = 249
    Cursor = crHandPoint  («ручка» - нужно нажимать фишки)
    ColCount = 4
    RowCount = 4  (запускаем игру в режиме «Флип-Флоп»)
    DefaultColWidth = 60
    DefaultRowHeight = 60  (размеры клеток поля достаточно боль-
шие, чтобы фишки выглядели «эффектно»)
    DefaultDrawing = False
    FixedCols = 0
    FixedRows = 0
    GridLineWidth = 0  (разметка сетки не нужна, её заменят линии
на рисунках с фишками)
    Options = []
    ScrollBars = ssNone
    OnDrawCell = dgPoleDrawCell
    OnMouseDown = dgPoleMouseDown
```


Свойства сетки *dgPole2*:

```
    Left = 260
    Top = 36
    Width = 129
    Height = 129
    Cursor = crArrow
    ColCount = 4
    RowCount = 4
    DefaultColWidth = 30
    DefaultRowHeight = 30
    DefaultDrawing = False
    FixedCols = 0
    FixedRows = 0
    GridLineWidth = 0
    Options = []
    ScrollBars = ssNone
    OnDrawCell = dgPole2DrawCell
    OnMouseDown = dgPole2MouseDown
```

Для удобства введём несколько глобальных *констант*:

```
const
  //макс. размеры поля:
  MAX_POLE_WIDTH= 7;
  MAX_POLE_HEIGHT= 7;
  //цвет фишек:
  WHITE=0;
  BLACK=1;
```

Хотя цвета фишек мы обозначили как «белый» и «чёрный», на самом деле они могут иметь любой цвет, важно только помнить, что *чёрные* фишки – это закрытые окна, а *белые* – открытые. Поэтому игра *Флип-Флоп* в варианте с фишками должна иметь конечную позицию, состоящую только

из *белых* фишек. Если у вас есть страстное желание украсить игру, то вместо фишек вы можете взять произвольные картинки (хотя бы и окна), на самой игре это никак не отразится. Вот только надо ли отвлекать игрока совершенно ненужными визуальными и звуковыми эффектами – это БОЛЬШОЙ вопрос!

Как обычно, для хранения информации о фишках на поле мы заведём два глобальных *массива*:

```
type
  TPole= array[0..MAX_POLE_WIDTH-1, 0..MAX_POLE_HEIGHT-1] of byte;
var
  //поля:
  masPole, masPole2: TPole;
```


Нам обязательно понадобится в игре *метка lblHod*, чтобы показывать число сделанных при решении головоломки ходов:

```
    Left = 268
    Top = 176
    Width = 86
    Height = 29
    Caption = 'Ход - 0'
    Font.Color = clRed
    Font.Height = -24
    Font.Style = [fsBold]
    ParentFont = False
    OnClick = lblHodClick
```

Иногда бывает полезно начать отсчёт ходов с какой-то определённой позиции, для этого достаточно *щёлкнуть* на метке:

```
//ОБНУЛИТЬ ХОДЫ
procedure TForm1.lblHodClick(Sender: TObject);
begin
  Hod:= 0;
  lblHod.Caption:=  'Ход - 0';
end;  //lblHodClick
```

При поиске решения также очень важно уметь сохранять и восстанавливать позицию на поле, а также переходить к предыдущему или последующему шагу в цепочке ходов. Для хранения текущей позиции мы введём переменную *Memory*:

```
type
  TPole= array[0..MAX_POLE_WIDTH-1, 0..MAX_POLE_HEIGHT-1] of byte;
  TMemory= Record
    //номер хода:
    hod: integer;
```

```
    //размеры полей:
    w, h: integer;
    //позиция на полях:
    PosL, PosR: TPole;
  end;

var
  //поля:
  masPole, masPole2: TPole;
  Memory: TMemory;
```


Чтобы не загромождать форму кнопками, добавим в проект всплывающее меню *PopupMenu1* (компонент TPopupMenu), которое будет появляться при нажимании *правой* кнопки мыши. Так оно выглядит в *Редакторе меню* (Рис. 5.9, слева) и в игре (Рис. 5.9, справа).

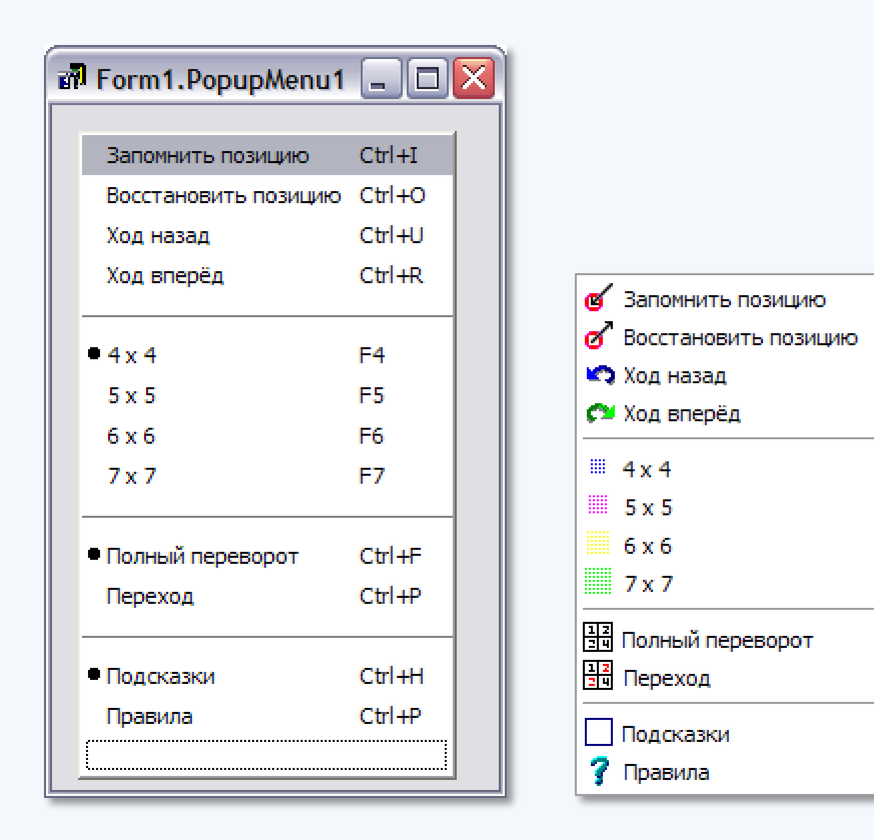


Рис. 5.9. Всплывающее меню

Лучше немного забежать вперёд и сразу создать *все* пункты меню, чем много раз возвращаться в *Редактор меню* и добавлять по одному пункту.

Чтобы каждый пункт меню предварял *значок*, загрузите в свойство *Bitmap* подходящую картинку. Часть из них вы можете взять из наших предыдущих программ, а другие нарисовать самостоятельно.

Первый пункт меню – *miMemoryIn* – отвечает за *сохранение* в памяти информации о текущей позиции на игровом поле:

```
Caption= 'Запомнить позицию'
ShortCut= Ctrl+I
OnClick = miMemoryInClick
```



```
//ЗАПОМНИТЬ ПОЗИЦИЮ
procedure TForm1.MemoryIn;
begin
  Memory.hod:= Hod;
  Memory.w:= dgPole.ColCount;
  Memory.h:= dgPole.RowCount;
  Memory.PosL:= masPole;
  Memory.PosR:= masPole2;
end; //MemoryIn

procedure TForm1.miMemoryInClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;
  MemoryIn;
end; //miMemoryInClick
```

Раз уж появилось упоминание о *статусе* игры, то следует рассказать о нём. Введём новый *тип*

```
TGameState= (gsWait, gsSolution);
```

и *переменную* этого типа

```
//состояние программы:
GameState: TGameState= gsWait;
```

В *первом* состоянии программа находится, когда ждёт ваших действий (выполнения хода, нажатия на кнопку, выбора пункта меню). Во *втором* – когда она ищет решение головоломки. Ясно, что в этом случае нужно отвергнуть все другие действия в программе (кроме прекращения самого поиска, иначе может возникнуть ситуация, когда нам придётся завершать программу негуманным способом).

Второй пункт меню – *miMemoryOut* – выполняет противоположную операцию - *восстанавливает* позиции на обоих полях:

```
Caption = 'Восстановить позицию'
ShortCut = Ctrl+O
OnClick = miMemoryOutClick
```

Так как размеры полей могли быть изменены, то нужно вывести *новые* поля, отметить в меню их действительные размеры и переместить форму в центр экрана (размеры формы будут автоматически подгоняться под размеры полей, поэтому ее размеры зависят от размеров сеток):

```
//ВОССТАНОВИТЬ ЗАПОМНЕННУЮ ПОЗИЦИЮ
procedure TForm1.miMemoryOutClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;

  with Memory do begin
    Prepare(dgPole, w, h);
    Prepare(dgPole2, w, h);
    masPole:= PosL;
    masPole2:= PosR;
    case w of
      4: mi4x4.Checked:= True;
      5: mi5x5.Checked:= True;
      6: mi6x6.Checked:= True;
      7: mi7x7.Checked:= True;
    end;
    form1.Position:= poScreenCenter;
  end;

  dgPole.Invalidate;
  dgPole2.Invalidate;
  Hod:= Memory.hod;
  lblHod.Caption:= 'Ход - ' + inttostr(hod);
end;  //miMemoryOutClick
```

Размеры полей задаются в *процедуре Prepare:*

```
//УСТАНОВИТЬ РАЗМЕР ЗАДАННОГО ПОЛЯ
procedure TForm1.Prepare(Pole: TDrawGrid; ColCount, RowCount: in-
teger);
var w,h, lw: integer;
begin
  //размер клетки в пикселях:
  w:= Pole.DefaultColWidth;
  h:= Pole.DefaultRowHeight;

  //толщина линий:
  lw:= Pole.GridLineWidth;

  //размеры игрового поля в клетках:
  Pole.ColCount:= ColCount;
  Pole.RowCount:= RowCount;

  //размеры в пикселях видимой части игрового поля:
  Pole.Width:= 3 + (w + lw)* ColCount+1;
  Pole.Height:= 3 + (h + lw)* RowCount+1;
```

```
  if pole= dgPole2 then
    pole.Left:= dgPole.Left+dgPole.Width+10;

  //установить метку, показывающую число ходов:
  lblHod.Left:= dgPole.Left+dgPole.Width+10;
  lblHod.Top:= dgPole2.Top+ dgPole2.Height+10;

  //обнулить ходы:
  Hod:=0;
  lblHod.Caption:= 'Ход - 0';
end;  //Prepare
```


Так как большинство действий одинаково для обоих полей, то для изменения их размеров мы используем *одну* процедуру. Нужно только учесть, что *горизонтальные* координаты полей, безусловно, отличаются.

Следующие два пункта меню мы пока пропустим, потому что их действие сейчас трудно объяснить. А далее в меню идут пункты, позволяющие выбрать один из доступных размеров полей. По умолчанию игровое поле имеет такие же размеры, как и в игре *Флип-Флоп*, то есть 4 на 4 клетки. Это естественно – неискушённый пользователь должен начинать с более лёгких заданий.

Пункт *mi4x4* устанавливает *минимальные* размеры полей:

```
      Tag = 4
      Caption = '4 x 4'
      Checked = True  (размер по умолчанию)
      GroupIndex = 1
      RadioItem = True  (одновременно можно выбрать только один
размер поля)
      ShortCut = F4
      OnClick = mi4x4Click
```

Аналогично действуют и другие пункты из этой группы (обратите внимание, что значение свойства *GroupIndex* у них равно 1, а *RadioItem* = *True;* это сделано для того, чтобы при выборе одного из пунктов все остальные выключались).

```
mi5x5:
      Tag = 5
      Caption = '5 x 5'
      GroupIndex = 1
      RadioItem = True
      ShortCut = F5
      OnClick = mi4x4Click

mi6x6:
      Tag = 6
```

```
        Caption = '6 x 6'
        GroupIndex = 1
        RadioItem = True
        ShortCut = F6
        OnClick = mi4x4Click

 mi7x7:
        Tag = 7
        Caption = '7 x 7'
        GroupIndex = 1
        RadioItem = True
        ShortCut = F7
        OnClick = mi4x4Click
```


Выбор всех пунктов меню мы будем обрабатывать в *одной* и той же процедуре *mi4x4Click*. А чтобы узнать, какие размеры полей нужно установить, присвоим свойству *Tag* соответствующее значение:

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЕЙ
procedure TForm1.mi4x4Click(Sender: TObject);
var
 i, j, n: integer;
begin
  if GameState= gsSolution then exit;

  //показать пункт меню как выбранный:
  (Sender as TMenuItem).Checked:= True;

  //размеры поля:
  n:= (Sender as TMenuItem).Tag;

  //новое задание:
  dgPole.RowCount:= n; dgPole.ColCount:= n;
  dgPole2.RowCount:= n; dgPole2.ColCount:= n;
  NewPlay;

  //показать на экране:
  Prepare(dgPole, n, n);
  Prepare(dgPole2, n, n);
  dgPole.Invalidate;
  dgPole2.Invalidate;
  form1.Position:= poScreenCenter;
end;  //mi4x4Click
```

После изменения размера полей текущее задание теряет всякий смысл, поэтому оба поля просто заполняются фишками одного цвета. О том, как начать новую игру, будет рассказано далее.

В игре *Флип-Флоп* требуется все чёрные фишки (ну хорошо, окна!) перевернуть на белую сторону. Для нас эта задача оказалась слишком

простой, поэтому мы придумали усложнённый вариант игры, в котором нужно перейти от одной «ненулевой» позиции к другой. Сохраним *обе* разновидности игры в нашей программе. Опишем их в новом *типе TGame:*

```
  //вид игры:
  TGame=(FullSalto, Perehod);
```

и заведём новую *переменную* для хранения выбранного вида игры:

```
var
  …
  GameState: TGameState= gsWait;
  Game: TGame;
```


По умолчанию мы будем играть в более простой вариант:

```
miFullSalto:
      Tag = 0
      Caption = 'Полный переворот'
      Checked = True    (игра по умолчанию)
      GroupIndex = 2
      RadioItem = True
      ShortCut = Ctrl+F
      OnClick = miFullSaltoClick

miPerehod:
      Tag = 1
      Caption = 'Переход'
      GroupIndex = 2
      RadioItem = True
      ShortCut = Ctrl+P
      OnClick = miFullSaltoClick
```

Выбор пункта меню в этом случае осуществляется точно так же, как и при изменении размеров поля:

```
//ВЫБРАТЬ ВИД ИГРЫ
procedure TForm1.miFullSaltoClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;

  (Sender as TMenuItem).Checked:= True;

  //вид игры:
  Game:= TGame((Sender as TMenuItem).Tag);

  //начать новую игру:
  NewPlay;
end; //miFullSaltoClick
```

Управившись с меню, мы возвращаемся к началу программы. При запуске приложения в процедуре создания формы мы устанавливаем её свойство *AutoSize* в *True*, чтобы форма *подгонялась* под размеры полей. Это позволит нам избежать «пустот» на форме, когда размеры полей 4 x 4 клетки, а форма рассчитана на поля максимального размера.

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //подстраиваем величину формы под размеры сеток:
  AutoSize:= True;

  //установить размеры полей по умолчанию:
  Prepare(dgPole, 4, 4);
  Prepare(dgPole2, 4, 4);

  //начать новую игру:
  NewPlay;
end; //FormCreate
```


Вся подготовка к новой игре производится в процедуре *NewPlay*. Здесь мы должны обнулить число ходов, а затем выставить фишки на игровом поле случайным образом, но так, чтобы задача не получилась слишком простой:

```
   repeat
    Randomize;
   //выставить на поле фишки случайным образом:
    for j:= 0 to dgPole.RowCount-1 do
      for i:= 0 to dgPole.ColCount-1 do
        masPole[i,j]:= Random(2);

    //подсчитать число нечётных рядов:
    nrow:=0;
    for j1:= 0 to dgPole.RowCount-1 do
      for i1:= 0 to dgPole.ColCount-1 do begin
        n:= 0;
        for j:= 0 to dgPole.RowCount-1 do
          for i:= 0 to dgPole.ColCount-1 do
            if (i=i1) or (j=j1) then
              n:=n+masPole[i,j];
        if odd(n) then inc(nrow);
      end;
  until nrow>5;  //повторять, пока не будет больше 5
  dgPole.Invalidate;
```

На поле-образец мы выставляем фишки в зависимости от вида игры. Если это *Флип-Флоп*, то мы просто ставим «белые» фишки, если *Xor* – поступаем аналогично первому полю:

```
    //выставить на поле фишки случайным образом:
    for j:= 0 to dgPole2.RowCount-1 do
      for i:= 0 to dgPole2.ColCount-1 do
        masPole2[i,j]:= Random(2);
```

Казалось бы, всё верно – ан нет: до этого мы занимались только полем 4 на 4 клетки, на котором *все* позиции разрешимы. Мы распространили наши познания на поля *любых* размеров и ошиблись - например, на поле 5 х 5 клеток некоторые позиции нельзя «обнулить». Разговор об этом пойдет дальше, но, чтобы не возвращаться к этой процедуре, мы изменим алгоритм выбора случайной позиции так, чтобы *любая* игровая позиция была решаемой. Сделать это достаточно просто – нужно выполнить несколько ходов из *нулевой* позиции. Ясно, что, поменяв порядок ходов на обратный, мы всегда сможем из заданной позиции вернуться к нулевой. Причём размер поля в данном случае не имеет никакого значения:


```
//ПОДГОТОВКА К НОВОЙ ИГРЕ
procedure TForm1.NewPlay;
var
 i, j: integer;
 x, y: integer;
begin
  //обнулить число ходов:
  Hod:= 0;
  lblHod.Caption:=  'Ход - 0';

  //сделать случайные ходы:
  randomize;
  //очистить левое поле:
  for j:= 0 to dgPole.RowCount-1 do
    for i:= 0 to dgPole.ColCount-1 do
      masPole[i,j]:= WHITE;

  //сделать случайные ходы:
  for i:= 1 to 7 + random(dgPole.ColCount*dgPole.RowCount+1-7) do
begin
    x:= Random(dgPole.ColCount); y:= Random(dgPole.RowCount);
    masPole:= DoMove(masPole, x, y);
  end;
  dgPole.Invalidate;

  //задать поле-образец:
  if Game= FullSalto then begin //- полный переворот
    //выставить на поле зелёные фишки:
    for j:= 0 to dgPole2.RowCount-1 do
      for i:= 0 to dgPole2.ColCount-1 do
        masPole2[i,j]:= WHITE
  end
  else begin //- переход к образцу
    //сделать случайные ходы:
```

```
    for i:= 1 to 7 + random(dgPole2.ColCount*dgPole2.RowCount+1-7)
do begin
      x:= Random(dgPole2.ColCount); y:= Random(dgPole2.RowCount);
      masPole2:= DoMove(masPole2, x, y);
    end;
  end;
  dgPole2.Invalidate;

  //запомнить начальную позицию:
  MemoryIn;
end;  //NewPlay
```


Чтобы выполнить ход, мы поступаем как и раньше, но теперь нам не удастся хранить информацию о поле в двухбайтовом слове, так как поля могут содержать более 16 клеток. Как вы помните, для этого мы ввели специальный тип *TPole*. Расход памяти под хранение информации о полях увеличился, но зато не нужно считать биты!

```
//ВЫПОЛНИТЬ ХОД
function TForm1.DoMove(arr: TPole; x, y: integer): TPole;
var i: integer;
begin
  //инвертировать заданную клетку:
  arr[x,y]:= abs(arr[x,y]-1);
  //инвертировать ряды:
  for i:= 0 to dgPole.RowCount-1 do //- вертикальный
    arr[x,i]:= 1- arr[x,i];
  for i:= 0 to dgPole.ColCount-1 do //- горизонтальный
    arr[i,y]:= 1- arr[i,y];
  result:= arr;
end; //DoMove
```

Остались последние два пункта меню – вывод *подсказок* и *помощи*. Подсказка заключается в том, что *нечётные* клетки (в них должен быть сделан очередной ход, если вы забыли) будут отмечены *синим квадратиком* (раньше мы использовали плюсик, но для настоящей игры он не годится по эстетическим соображениям).

```
miHint:
      Caption = 'Подсказки'
      Checked = True  (по умолчанию подсказки выводятся на экран)
      GroupIndex = 3
      RadioItem = True
      ShortCut = Ctrl+H
      OnClick = miHintClick
```

При переключении режима подсказок оба поля обновляются, чтобы показать или убрать синие квадраты:

```
//ПОДСКАЗКИ
procedure TForm1.miHintClick(Sender: TObject);
begin
  miHint.Checked:= not miHint.Checked;
  dgPole.Invalidate;
  dgPole2.Invalidate;
end;  //miHintClick
```

```
miHelp:
      Caption = 'Правила'
      GroupIndex = 4
      ShortCut = Ctrl+P
```




После того как вы закончите программировать игру, напишите сами «справочное пособие» для начинающих пользователей!

Как я уже отмечал выше, фишки могут быть *любого* цвета, а не только белыми и чёрными (они названы так, чтобы не было путаницы при изменении цвета настоящих фишек). В данном случае роль чёрных фишек играют *красные*, а белых – *зелёные*. Чтобы игровое поле выглядело ещё привлекательнее, можно добавить *фон*. Размер клеток поля 60 х 60 пикселей, такой же величины должны быть и растровые картинки. Кнопки можно нарисовать в любом графическом редакторе, но лучше использовать специализированные редакторы для *Web*-графики, например *Xara WebStyle*. И последнее: не забудьте по контуру рисунка выполнить чёрную рамку (Рис. 5.10).

Рис. 5.10. Цветные фишки

Для хранения этих картинок установите на форме два компонента *TImage* TImage и загрузите в свойство *Picture* картинки с диска.

```
imgWhite:
    Width = 60
    Height = 60
    AutoSize = True
    Visible = False

imgBlack:
    Width = 60
    Height = 60
```

```
    AutoSize = True
    Visible = False
```

При выводе сетки на экран вызывается *процедура dgPoleDrawCell*, в которую и нужно поместить все операции по отрисовке каждой клетки поля в зависимости от того, какая фишка в ней находится. В нужную ячейку поля копируется соответствующая картинка, а затем, если необходимо, в нечётной клетке рисуется синий квадрат:

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var
  i, j, n, m: integer;
  r, dr: TRECT;
begin
  //фишка в ячейке:
  n:= masPole[ACol, ARow];

  //размеры картинок:
  r:= Bounds(0, 0, 60, 60);
  dr:= Bounds(Rect.Left, Rect.Top, 60, 60);

  case n of
      WHITE: //- белая фишка
        dgPole.canvas.CopyRect(dR, imgWhite.Canvas, R);
      else   //- чёрная фишка
        dgPole.canvas.CopyRect(dR, imgBlack.Canvas, R);
  end;

  //выводить подсказки для нечётных позиций?
  m:= dgPole.ColCount-1;
  if miHint.Checked then begin // - выводить подсказки
    //нечётная клетка?
    n:= 0;
    for j:= 0 to m do
      for i:= 0 to m do
        if (i=ACol) or (j=ARow) then begin
          n:=n+masPole[i,j];
        end;
    if odd(n) then begin // - нечётная!
      //нарисовать контурный квадрат:
      InflateRect(dr,-3,-3);
      dgPole.Canvas.Brush.Style:= bsClear;
      //цвет линий:
      dgPole.Canvas.Pen.Color:= clBlue;
      dgPole.Canvas.Pen.Width:= 1;
      dgPole.Canvas.Rectangle(dr.left, dr.top, dr.right,
dr.bottom);
    end;
```

```
  end;
end;  //dgPoleDrawCell
```

Точно так же выводятся на экран и клетки поля-образца. Есть только одно различие: клетки этого поля в 2 раза меньше размера картинок, поэтому метод канвы *CopyRect* не подходит, ему на смену приходит более мощный метод *StretchDraw*, который «умеет» подгонять картинку под любые размеры:

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);
var
  i, j, n, m: integer;
  dr: TRECT;
begin
  n:= masPole2[ACol, ARow];
  dr:= Bounds(Rect.Left, Rect.Top, Rect.Right-
Rect.Left,Rect.Bottom-Rect.Top);
  case n of
      WHITE: //- белая фишка
        dgPole2.canvas.StretchDraw(dR, imgWhite.Picture.Graphic);
      else   //- чёрная фишка
        dgPole2.canvas.StretchDraw(dR, imgBlack.Picture.Graphic);
  end;

  //выводить подсказки для нечётных позиций?
  m:= dgPole2.ColCount-1;
  if miHint.Checked then begin
    n:= 0;
    for j:= 0 to m do
      for i:= 0 to m do
        if (i=ACol) or (j=ARow) then begin
          n:=n+masPole2[i,j];
        end;
    if odd(n) then begin
     InflateRect(dr,-2,-2);
     dgPole2.Canvas.Brush.Style:= bsClear;
     dgPole2.Canvas.Pen.Color:= clBlue;
     dgPole2.Canvas.Pen.Width:= 1;
     dgPole2.Canvas.Rectangle(dr.left, dr.top, dr.right,
dr.bottom);
    end;
  end;
end;  //dgPole2DrawCell
```


Вполне уместно сразу рассмотреть и вторую процедуру для игрового поля. При нажатии в какой-либо клетке *левой* кнопки мыши выполняется ход, то есть «переворачиваются» фишки в соответствующей колонке и строке.

Как это делается, мы уже разобрали. Если вы захотите выставить на поле *свою* позицию, то при нажатии кнопки мыши удерживайте клавишу *Shift* (правая кнопка мыши уже занята под вызов меню). В этом случае изменит цвет только *одна* фишка.

```
//ПЕРЕВЕРНУТЬ ФИШКУ ИЛИ СДЕЛАТЬ ХОД
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  i, j: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  //нажата левая кнопка мыши и клавиша Shift -
  //инвертируем цвет клетки:
  if (ssLeft in shift) and (ssShift in shift ) then begin
    if masPole[ACol,ARow]= 1 then masPole[ACol,ARow]:= 0
    else masPole[ACol,ARow]:= 1;
    dgPole.Invalidate;
  end
  //нажата левая кнопка мыши без клавиши Shift -
  //делаем ход - инвертируем ряды:
  else if ssLeft in shift then begin
    inc(Hod); lblHod.Caption:= 'Ход - ' + inttostr(Hod);
    for j:= 0 to dgPole.RowCount-1 do
      for i:= 0 to dgPole.ColCount-1 do
        if (i=ACol) or (j=ARow) then
        //инвертировать клетку:
        masPole[i,j]:= abs(masPole[i,j]-1);
    dgPole.Invalidate;
    //запомнить ход:
    Moves[Hod].x:= ACol;
    Moves[Hod].y:= ARow;
    //проверить, не решена ли задача:
    if IsReady then Ready;
  end;
end;  //dgPoleMouseDown
```

Чтобы иметь возможность брать ходы назад, следует вести «протокол» - заполнять массив *Moves* координатами каждого хода. Объявите его в разделе глобальных переменных. Там же разместите и *переменную* для хранения числа сделанных ходов:

```
var
  …
  Game: TGame;
  //запись ходов:
  Moves: array[0..999] of TPoint;
```

```
  //номер хода:
  Hod: integer= 0;
```

После каждого хода нужно проверять, не решена ли задача. Доверим это *функции IsReady*. Если найдётся *хотя бы одна* фишка на *игровом* поле, отличная от фишки в соответствующей клетке *поля-образца*, то радоваться, увы, рано, в противном случае мы дошли до конечной позиции и имеем полное право поздравить себя с победой:


```
//ПРОВЕРИТЬ, НЕ ПОЛУЧИЛАСЬ ЛИ КОНЕЧНАЯ ПОЗИЦИЯ
function TForm1.IsReady: Boolean;
var
  i, j: integer;
begin
  Result:= True;
  for j:= 0 to dgPole.RowCount-1 do
    for i:= 0 to dgPole.ColCount-1 do
      if masPole[i,j]<> masPole2[i,j] then
        begin Result:= False; exit end
end; //IsReady
```

Поздравление мы оформим без изысков (Рис. 5.11).



Рис. 5.11. Скромненько, но со вкусом!

За основу примем компонент TPanel *Panel1*:

```
Width = 196
Height = 45
AutoSize = True
BevelInner = bvLowered
BorderStyle = bsSingle
Color = clBlack
Visible = False,
```

на котором расположим *метку Label1*:

```
Left = 2
Top = 2
Width = 188
Height = 37
Caption = 'Получилось!'
Font.Color = clRed
Font.Height = -32
Font.Style = [fsBold]
```

В *процедуре Ready* мы показываем панель в середине формы, обнуляем счётчик (объявите глобальную переменную *rep: integer= 0;*) и включаем *таймер*:

```
//ЗАДАЧА РЕШЕНА!
procedure TForm1.Ready;
begin
  panel1.Left:= (Form1.Width- panel1.Width) div 2;
  panel1.Top:= (Form1.Height- panel1.Height) div 2;
  panel1.Visible:= true;
  rep:= 0;
  timer1.Enabled:= true;
end; // Ready
```


Настройте компонент *TTimer1* так, чтобы он срабатывал примерно 2 раза в секунду:

```
    Interval = 400
    OnTimer = Timer1Timer
```

Через установленный промежуток времени будет выполняться следующая процедура:

```
//МИГАЮЩАЯ НАДПИСЬ
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  inc(rep);
  if odd(rep) then label1.Font.Color:= clRed
  else label1.Font.Color:= clGreen;
  if rep>15 then begin
    timer1.Enabled:= false;
    panel1.Visible:= false;
  end;
end;  // Timer1Timer
```

В ней цвет надписи изменяется с красного на зелёный и наоборот. Через 15 миганий таймер выключится, а панель исчезнет с экрана.

Так как на правом поле находится *конечная* позиция, то можно позволить себе устанавливать её по собственному желанию, а вот делать ходы на этом поле не нужно, поэтому процедура-обработчик нажатия кнопки мыши будет проще, чем для игрового поля:

```
//ПЕРЕВЕРНУТЬ ФИШКУ НА ПОЛЕ-ОБРАЗЦЕ
procedure TForm1.dgPole2MouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
```

```
begin
  //координаты мыши:
  dgPole2.MouseToCell(x,y,ACol,ARow);
  //нажата левая кнопка мыши - инвертируем цвет клетки:
  if ssLeft in shift then begin
    if masPole2[ACol,ARow]= 1 then masPole2[ACol,ARow]:= 0
    else masPole2[ACol,ARow]:= 1;
    dgPole2.Invalidate;
  end
end;  // dgPole2MouseDown
```


Теперь у нас имеется всё необходимое для того, чтобы решать задачи наподобие тех, что мы рассматривали в программе *Флип-Флоп*. Как вы помните, нахождение *короткого* перехода от одной ненулевой позиции к другой было оставлено для самостоятельного анализа. Конечно, вы догадались, что лишние ходы в решении возникают из-за клеток с плюсиками, одинаково расположенными на *обоих* полях (обе клетки *нечётные*). Выполнив *два* хода в одну и ту же клетку, мы только продлили решение задачи, но не изменили его. Отсюда следует правило: *для перехода из одной позиции в другую следует ходить на игровом поле в те нечётные клетки, которым соответствуют чётные клетки на поле-образце, и, наоборот, в те чётные клетки, которым там соответствуют нечётные*.

Технически это выглядит так. Мы последовательно перебираем пары соответствующих клеток обоих полей. Если ровно одна из них нечётная, то мы делаем ход в эту клетку игрового поля. Если они обе чётные или обе нечётные, то такие клетки мы пропускаем и переходим к следующей паре. Закончив обход всего поля (или раньше, если повезёт), мы неминуемо решим задачу.

Для примера рассмотрим конкретную задачу (Рис. 5.12).

На рисунках *серым* цветом выделены *нечётные* клетки на *обоих* полях. Ходы нужно выполнять только в те клетки игрового поля, которые отмечены *крестиком*. Всего для решения задачи потребуется 9 ходов.

Легко проверить, что, действуя аналогично, вы справитесь с *любой* задачей на поле 6 х 6 клеток (а также на любом другом квадратном с *чётным* количеством клеток; именно поэтому в программе отсутствует режим 8 х 8 клеток – начинающим игрокам задачи на таком большом поле будут не по зубам, а для знатоков в них нет ничего нового).



Рис. 5.12. Как решить задачу

Сюрпризы начинаются на поле 5 x 5 клеток. Куда бы вы ни сделали *первый* ход из нулевой позиции, вы получите не одну, а сразу *несколько* нечётных клеток (Рис. 5.13).

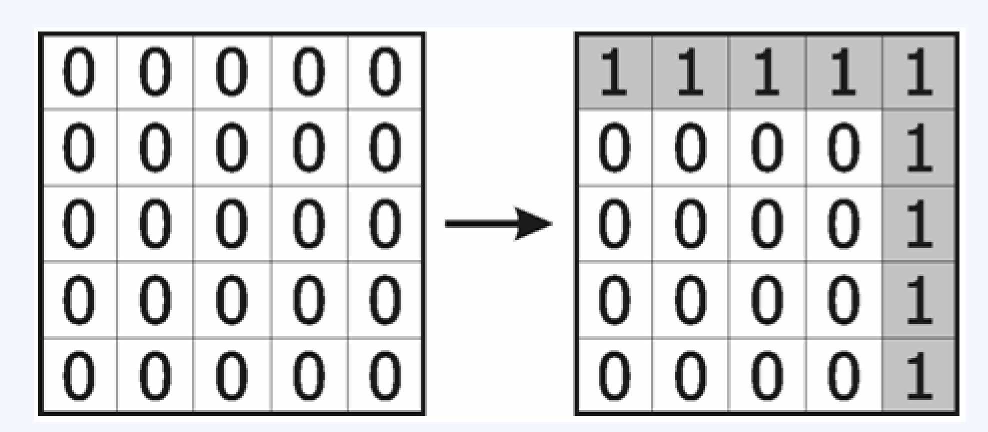


Рис. 5.13. Приключения начинаются!

Причём только одна из них «правильная», а все остальные не ведут к решению задачи. Ситуация становится ещё более напряжённой после *второго* хода (Рис. 5.14).

1	1	1	1	1
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1

→

1	1	0	1	1
0	0	1	0	1
1	1	1	1	0
0	0	1	0	1
0	0	1	0	1

Рис. 5.14. Усугубляем!

А творится это «безобразие» потому, что теперь в каждой колонке и строке *нечётное* количество клеток, а это значит – все клетки с единицей будут нечётными, а с нулём – чётными. Это легко проверить для позиции, возникающей из нулевой после *первого* хода (впрочем, все дальнейшие рассуждения верны и для этой позиции). Если *второй* ход сделать в клетку с 0 (она чётная), то в ней появится 1, при этом изменятся числа в 9 клетках соответствующей колонки и строки. До второго хода в них было чётное количество единиц (ходим в чётную клетку!) и нечётное количество нулей (всего чисел 9 – нечётное число), а после него будет нечётное количество единиц, так что клетка превратится в *нечётную*. Можете проверить, что, сделав ход в нечётную клетку (с единицей), вы получите *чётную* клетку. Естественно, третий ход только подтвердит эту печальную традицию.

Так как чётность клеток *изменяется* при каждом ходе, то невозможно определить, в какие из них были сделаны предыдущие ходы. Зато мы теперь можем легко показать существование *неразрешимых* позиций. Выставим на нулевое поле одну единицу. Например, в левый верхний угол (Рис. 5.15).

1	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Рис. 5.15. Первый ход – и сразу в угол!

В этой позиции нечётной оказывается не только клетка с единицей, но и ещё 8 клеток с нулями! Но мы доказали, что во всех позициях, возникающих из нулевой, клетки с нулями могут быть исключительно *чётными*. Таким образом, указанная позиция не может получиться из нулевой и, соответственно, не может быть сведена к ней никакими ходами. Вы можете легко найти *множество* неразрешимых позиций, если запустите программу, очистите левое поле (все фишки станут *зелёными*), установите его размеры 5 на 5 клеток и будете выставлять *красные* фишки левой кнопкой мыши с шифтом. Если в какой-либо позиции хотя бы одна красная фишка станет *чётной* (не будет выделена синим квадратом) или хотя бы одна зелёная нечётной (будет выделена) – значит, вы получили *неразрешимую* позицию. Смело заключайте пари, что никто эту задачу не решит (то есть не сможет по правилам игры перевернуть все фишки зелёной стороной вверх). Так развлекался ещё сам Лойд со своими *пятнашками* – почему бы и вам не пошутить (разумеется, речь идёт о шуточном пари!)?

Вследствие того, что при случайном выставлении фишек на поле размером 5 на 5 клеток нередко будут возникать *неразрешимые* позиции, нам и пришлось изменить алгоритм задания новой позиции в процедуре *NewPlay*. Произвольное выставление фишек мы заменили случайными ходами, которые, безусловно, никак не могут привести к неразрешимым позициям. Это верно и для *второго* варианта игры, когда совершается переход от одной ненулевой позиции к другой ненулевой позиции, так как и в этом случае конечная позиция возникает как результат случайных ходов на поле-образце.

Итак, любая задача в *нашей* игре (имеется в виду поле 5 х 5) решается, вопрос только в том – *как* следует ходить, чтобы её решить. Нетрудно убедиться, что ходы в нечётные клетки (с красными фишками), в отличие от поля 4 х 4, к успеху не ведут – число нечётных клеток хаотически изменяется, но отнюдь не убывает! Всё, что мы *нажили непосильным трудом*, *пошло прахом* – нужно искать *новые* подходы к решению задач на поле 5 х 5 клеток. Или вы уже смекнули, как необходимо действовать в этом случае? Если это так, то можно только позавидовать вашей прозорливости. А всех остальных я приглашаю на следующий тур наших изысканий.

Попробуем воспользоваться нашим умением решать задачи на *чётных* полях. Выделим на поле 5 х 5 клеток квадрат размером 4 х 4 клетки. Сделать это можно четырьмя разными способами, но нам удобнее совместить оба поля верхними левыми углами. Для примера рассмотрим реальную задачу, предложенную нашей программой.

Мы будем выделять нечётные клетки только для *малого* квадрата, исключая остальные клетки *большого* квадрата при подсчёте единиц (красных фишек) в соответствующих колонках и строках (Рис. 5.16).

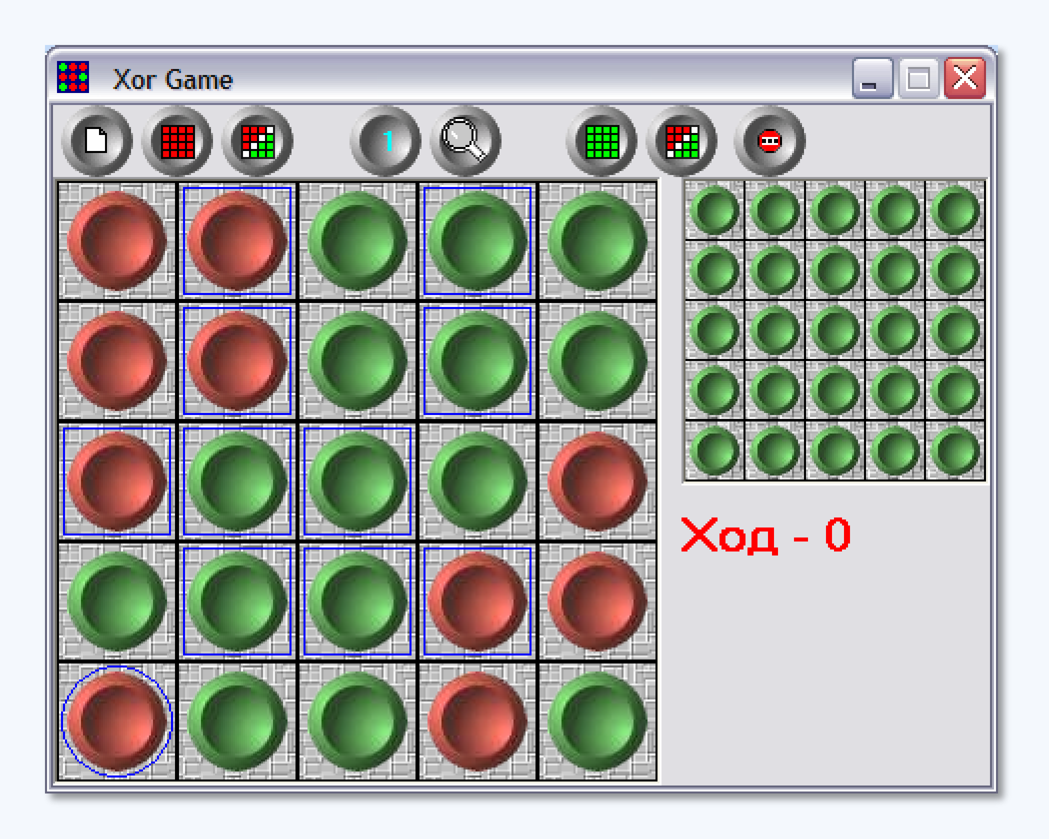


Рис. 5.16. Изыскательские работы продолжаются!

Если этого не сделать, то решение квадрата 4 x 4 будет затруднено (придётся нам самим отыскивать в нём нечётные клетки). Разумеется, чётность клеток при таком подсчёте изменится, что и отражено на рисунках.

Само решение задачи на поле 4 на 4 клетки не должно вызвать у вас никаких затруднений. Как и прежде, ходить нужно в *нечётные*, выделенные клетки. Ясно видно, что потребуется 10 ходов, так как выделено именно столько клеток. В результате получится такая позиция (Рис. 5.17).

0	0	0	0	?
0	0	0	0	?
0	0	0	0	?
0	0	0	0	?
?	?	?	?	0

Рис. 5.17. Начинаем с нулей!

Число в *правом нижнем* углу останется тем же самым, а все остальные числа по периметру поля могут измениться (решая задачу на доске 4 х 4 мы обязаны инвертировать полные ряды длиной 5 клеток, иначе мы не сможем распространить наше решение на полный квадрат 5 х 5), поэтому числа в этих клетках заменены *знаком вопроса*. Попытаемся определить, какие числа за ними скрываются.

Так как в левом верхнем углу находится 0 (*чётная* клетка), то последние числа в левой колонке и в верхней строке либо оба нули, либо оба единицы (в противном случае упомянутый нуль будет стоять в нечётной клетке, что невозможно). Идя от верхнего левого угла вниз по диагонали и рассуждая по пути аналогично, мы придём к выводу, что периферийные клетки (исключая угловую с нулём) содержат: либо все единицы, либо все нули.

Пусть это будут *единицы*, тогда каждая из периферийных клеток должна быть *чётной*, так как в последней колонке и нижней строке будет по 4 единицы (Рис. 5.18).

0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
0	0	0	0	1
1	1	1	1	0

Рис. 5.18. Единицы?

Но такая позиция *невозможна*, поэтому во всех периферийных клетках должны находиться *нули*, и задача решена (Рис. 5.19).

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Рис. 5.19. Нули!

Если бы в правом нижнем углу стояла *единица*, то, сделав ход в эту клетку, мы пришли бы к рассмотренному нами решению. Но можно поступить иначе: сначала решить квадрат 4 х 4, тогда во всех периферийных клетках останутся *единицы* (доказательство этого очевидного факта проводится аналогично предыдущему случаю) и потребуется ещё *один* ход для окончания решения.

Пользуясь этим алгоритмом, вы легко справитесь с любой задачей и на поле 5 х 5 клеток. Правда, процедуры рисования фишек *dgPoleDrawCell* и *dgPole2DrawCell* не умеют выделять на нём квадрат 4 х 4. Прежде чем мы улучшим эти процедуры, нужно ещё решить, что делать с полем 7 х 7 клеток. Горячие головы, наверное, уже успели подумать, что и на нём следует выделить квадрат 4 на 4 клетки. А что вы прикажете делать с оставшимися *тремя* рядами периферийных клеток? И не надейтесь, что они так же благополучно «обнулятся», как и на поле 5 х 5 клеток. И тем не менее всё просто – вместо поля 4 х 4 нужно взять поле 6 х 6, вот тогда останется *один* периферийный слой фишек – со всеми вытекающими отсюда последствиями (если вы захотите ввести в программу режим 9 х 9 клеток, то сами знаете, какое нужно взять для него меньшее поле).

Добавьте в упомянутые процедуры такой код (выделенные строки):

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
…

  //выводить подсказки для нечётных позиций?
  if (dgPole.ColCount= 5) and ((ACol>3) or (ARow>3)) then exit;
  if (dgPole.ColCount= 7) and ((ACol>5) or (ARow>5)) then exit;
  if dgPole.ColCount= 5 then m:= 3
  else if dgPole.ColCount= 7 then m:= 5
  else m:= dgPole.ColCount-1;
…
end;  //dgPoleDrawCell

//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);
…
  //выводить подсказки для нечётных позиций?
  if (dgPole2.ColCount= 5) and ((ACol>3) or (ARow>3)) then exit;
  if (dgPole2.ColCount= 7) and ((ACol>5) or (ARow>5)) then exit;
  if dgPole2.ColCount= 5 then m:= 3
  else if dgPole2.ColCount= 7 then m:= 5
  else m:= dgPole2.ColCount-1;
```

```
…
end;  //dgPole2DrawCell
```

Порадовавшись своим успехам, давайте всё же задумаемся, а *всегда* ли наш алгоритм решения задач на *нечётных* полях находит *самое короткое* из всех возможных решений, как в случае чётных полей. Можно вручную перебирать ходы во множестве задач, чтобы проверить наши сомнения. Но это процесс трудоёмкий и нетворческий. Поэтому мы напишем процедуру поиска кратчайшего перехода между двумя позициями, взяв за основу ту, что сослужила нам хорошую службу в программе *Флип-Флоп*. Но мы изменим её так, чтобы она находила лучшее решение *самостоятельно*, без нашего участия. Задав такое число ходов, которого заведомо хватит для решения задачи, мы всё остальное поручим процедуре поиска. Например, в разобранной задаче для решения потребуется не более 10 ходов, поэтому разумно ограничить глубину поиска именно этим числом. Найдя такое решение, процедура продолжит поиск более коротких. И если они существуют, то обязательно будут обнаружены. Так, в нашем случае довольно *шести* ходов (нумерация клеток в этот раз более привычная для непрограммистов – начинается с *единицы*):


```
1.          2 - 3.
2.          5 - 3.
3.          3 - 4.
4.          1 - 5.
5.          3 - 5.
6.          5 - 5.
```

Кроме указанного, имеется ещё 3 варианта решения задачи такой же длины. Таким образом, мы делали 4 *лишних* хода. Это свидетельствует о том, что наш алгоритм не позволяет находить лучшее решение (по крайней мере, для некоторых задач). Причину этого «дефекта» несложно обнаружить, если внимательно посмотреть на последовательность ходов, выданную процедурой. В ней присутствуют ходы в периферийные клетки, которых мы избегали.

Длина решения на поле 4 х 4 определяется числом *нечётных* клеток на нём и может быть уменьшена хорошим ходом в *периферийную* зону. Для поля 5 х 5 клеток нужно проверить 8 клеток (угловая не в счёт), сделав в каждую из них ход и подсчитав нечётные клетки на поле 4 х 4. Для нашей задачи следует выбрать ход в нижний левый угол, что уменьшит число нечётных клеток на поле 4 х 4 до четырёх. При этом в правом нижнем углу будет стоять уже не 0, а 1, так что для решения задачи понадобится 6 ходов:

```
1.        1 - 5.
2.        3 - 1.
3.        3 - 2.
```

```
4.       1 - 3.
5.       4 - 3.
6.       5 - 5.
```

Это один из вариантов (второй), выданный процедурой поиска, которая в программе выглядит так:

```
//НАЙТИ ПЕРЕХОД ОТ ОДНОЙ ПОЗИЦИИ К ДРУГОЙ
procedure TForm1.Solution(nStep: integer);
//nStep2 - количество шагов (ходов)
label
  nextHod, nextCell;
var
  Hod: integer;
  nCells: integer;
  nCell: array[0..99] of integer;
  Pos: array[0..99] of TPole;
  Move: array[0..99] of integer;
  x, y: byte;
  w,h: integer;
  variant: integer;
  arr1, arr2: TPole;
  s: string;

  //ПРОВЕРИТЬ, НЕ ПОЛУЧИЛАСЬ ЛИ КОНЕЧНАЯ ПОЗИЦИЯ
  function IsOK: Boolean;
  var
    i, j: integer;
  begin
    Result:= True;
    for j:= 0 to h-1 do
      for i:= 0 to w-1 do
        if Pos[Hod][i,j]<> arr2[i,j] then
          begin Result:= False; exit end
  end;

  //записать ходы в протокол:
  procedure SavePos;
  var
    n, i, j: integer;
    x, y: integer;
    s: string;
  begin
    frmProtokol.listbox1.items.add('');
    for n:= 1 to Hod do begin
      s:='';
      x:= Move[n] mod w; y:= Move[n] div w;
      frmProtokol.listbox1.items.add('Шаг= '+ inttostr(n) +
'  Ход= '+ inttostr(x+1)+ ' ' +inttostr(y+1));

      for j:= 0 to h-1 do begin
        for i:= 0 to w-1 do
```

```
          if Pos[n][i,j]= BLACK then //- клетка фигуры
            s:= s+'1'
          else
            s:= s+'0';      //- клетка "фона"
        frmProtokol.listbox1.items.add(s);
        s:= '';
      end;
    end;
  end;

begin
  //очистить протокол:
  frmProtokol.listbox1.Clear;
  //скопировать массивы полей:
  arr1:= masPole; arr2:= masPole2;
  //размеры полей:
  w:= dgPole.ColCount; h:= dgPole.RowCount;
  //число найденных решений:
  variant:= 0;
  //число клеток на поле:
  nCells:= w*h;
  //глубина поиска:
  Hod:= 0;
  //начальная позиция:
  Pos[0]:= arr1;
  nCell[0]:= -1; Move[0]:= -1;
//делаем следующий ход:
nextHod:
  application.ProcessMessages;
  if flgStop= True then begin flgStop:= False; exit end; //-
остановить поиск
  Inc(Hod); nCell[Hod]:= nCell[Hod-1];  Pos[Hod]:= Pos[Hod-1];
//переходим к следующей клетке поля:
nextCell:
  inc(nCell[Hod]);
  if nCell[Hod]>= nCells then begin//-прошли всё поле
    dec(Hod);
    if Hod< 1 then begin
      if variant= 0 then
        s:= 'Задача решений не имеет!'
      else begin
        frmProtokol.Show;
        s:= 'Найдены все варианты - ' + inttostr(variant);
      end;
      ShowMessage(s);
      exit;
    end;
    Goto nextCell
  end;

  x:= nCell[Hod] mod w; y:= nCell[Hod] div w;
  //выполнить ход и запомнить позицию:
  Pos[Hod]:=DoMove(Pos[Hod-1], x, y);
```

```
  //запомнить ход:
  Move[Hod]:= nCell[Hod];

  if IsOK then begin //- нашли!
    //не искать длинных вариантов:
    if hod< nStep then begin
      nStep:= hod;
      variant:= 0;
    end;
    inc(variant);
    frmProtokol.listbox1.items.add('Вариант - '+ intto-
str(variant)+ '  Длина - ' + inttostr(hod));
    frmProtokol.listbox1.items.add('----------------------');
    savepos;
  end;

  if Hod< nStep then goto NextHod;
  goto nextCell
end;  // Solution
```


Для вывода найденных вариантов ходов мы добавим к проекту новую *форму* – *frmProtokol* (Рис. 5.20).

```
  Width = 205
  Height = 321
  BorderIcons = [biSystemMenu]
  Caption = '  Варианты'
```

с компонентом *список* - TListBox *ListBox1*:

```
  Left = 0
  Top = 0
  Width = 197
  Height = 293
  Font.Charset = RUSSIAN_CHARSET
  Font.Name = 'Courier New'
```

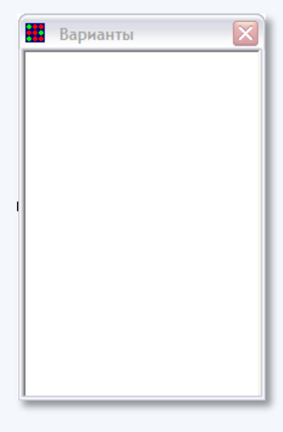


Рис. 5.20. Контора пишет!

В модуле формы добавьте строку:

```
implementation
uses ProtokolUnit;
```

Код модуля формы протокола (набирать вам его не придётся):

```
unit ProtokolUnit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls;

 type
  TfrmProtokol = class(TForm)
    ListBox1: TListBox;

  private
    { Private declarations }

  public
    { Public declarations }
  end;

var
  frmProtokol: TfrmProtokol;

implementation

{$R *.DFM}

end.
```


Если при поиске решения вы зададите *большую* глубину перебора, то этот процесс может и затянуться. Поэтому необходимо предусмотреть возможность выхода из процедуры поиска. В этих двух строках проверяется, не приняла ли переменная *flgStop* значение *True*. И если это так, поиск превращается:

```
  application.ProcessMessages;
  if flgStop= True then begin flgStop:= False; exit end; //-
остановить поиск
```

Объявите эту переменную в разделе глобальных переменных:

```
flgStop: boolean= false;
```

Чтобы остановить решение, нужно нажать кнопку со значком *СТОП*. Прежде чем написать процедуру для обработки нажатия на кнопку, мы поговорим немного о самих *кнопках*.

Выберем для нашей программы кнопки подходящей формы. Давайте сделаем их в том же стиле, что и фишки, чтобы они гармонировали друг с другом. Среди стандартных кнопок *Turbo Delphi* вы таких не найдёте, но ведь мы можем и сами сделать *круглые* кнопки! Воспользуемся тем же приёмом, что и в *тетрисе*, но сделаем их более удобными для работы.

Роль кнопок будет играть компонент *TImage*, в который нужно загрузить изображение соответствующей кнопки. Как их подготовить, я уже рассказывал, хотел бы только напомнить, что изображения кнопок должны быть в формате *BMP* на *белом* фоне.

Начнём с *последней* кнопки на форме, которая и будет останавливать поиск решений (Рис. 5.21).

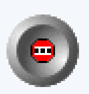


Рис. 5.21. Стоповая кнопка

Установите на форме компонент *imgStop* типа *TImage* и задайте ему свойства:

```
Left = 340
Top = 0     (это свойство для всех кнопок одинаково)
Width = 36
Height = 36   (размеры всех кнопок 36 x 36 пикселей)
Cursor = crHandPoint         (и курсор – тоже)
Hint = 'Остановить решение'
ParentShowHint = False
ShowHint = True
AutoSize = True
Picture – загрузите нужную картинку.
Transparent = True
OnMouseDown = imgStopMouseDown
OnMouseUp = imgStopMouseUp
```

Потребуется и *вторая* картинка для этой кнопки, она будет заменять первую, когда кнопка находится в *нажатом* положении (Рис. 5.22).

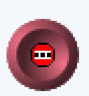


Рис. 5.22. Полный стоп!

За основу картинок для всех кнопок можно взять те же фишки, но меньшего размера и другого цвета (Рис. 5.23).

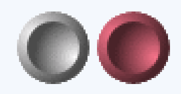


Рис. 5.23. Заготовки для кнопок

В каком-нибудь графическом редакторе на них нужно наложить значки-пиктограммы, как это принято в современных приложениях.

Поскольку нам придётся выводить на кнопку *два* разных изображения, то их где-то следует хранить. Например, можно приспособить для этого тот же компонент *TImage*. Тогда на каждую кнопку придётся ещё пара этих компонентов. Так, жизнедеятельность нашей кнопки обеспечивают *Image15* и *Image16*. Их свойства мало чем отличаются от компонента с кнопкой:

```
    Width = 36
    Height = 36
    AutoSize = True
    Picture – картинка с кнопкой в нормальном (Image15) и нажатом
(Image16) состоянии
    Transparent = True
    Visible = False
```

При нажатии на кнопку одна картинка на ней заменяется другой:

```
//ОСТАНОВИТЬ ПОИСК РЕШЕНИЙ
procedure TForm1.imgStopMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameState<> gsSolution then exit;
  imgStop.Picture.Assign(Image16.Picture);
end;  // imgStopMouseDown
```

А при отпускании возвращается первоначальная картинка и устанавливается флаг выхода из процедуры поиска:

```
procedure TForm1.imgStopMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameState<> gsSolution then exit;
  imgStop.Picture.Assign(Image15.Picture);
  flgStop:= True;
end;  // imgStopMouseUp
```

Если вы захотите сразу же и выйти из программы, то в *процедуре FormClose* также установите флаг выхода из процедуры поиска (на всякий случай – вдруг в это время программа углубилась в поиск перехода от одной позиции к другой):

```
//ЗАКРЫТЬ ФОРМУ
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAc-
tion);
begin
  flgStop:= True;
end; // FormClose
```


Глубина поиска решения задаётся *кнопкой imgStep*:

```
    Left = 148
    Hint = ' Глубина перебора'
    OnMouseDown = imgStepMouseDown
    OnMouseUp = imgStepMouseUp
```

Картинки для неё хранятся в компонентах *Image11* и *Image12*. Это просто кнопка, без картинок, так как на ней будет выводиться максимальное число ходов при поиске решения. Поэтому поместите на кнопку метку *lblStep*, которая и будет показывать это число:

```
    Left = 164
    Top = 10
    Width = 9
    Height = 16
    Cursor = crHandPoint (дублируем свойства самой кнопки, ведь
мышка во время нажатия кнопки может оказаться и на метке)
    Hint = 'Глубина перебора'
    Caption = '1'
    Font.Color = clAqua
    Font.Height = -13
    Font.Name = 'MS Sans Serif'
    Font.Style = [fsBold]
    ShowHint = True
    Transparent = True
    OnMouseDown = imgStepMouseDown
    OnMouseUp = imgStepMouseUp
```

При каждом нажатии на кнопку глубина перебора будет увеличиваться или уменьшаться на единицу в допустимых пределах:

```
//ЗАДАТЬ ГЛУБИНУ ПЕРЕБОРА
procedure TForm1.imgStepMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  //заменить картинку:
```

```
  imgStep.Picture.Assign(Image12.Picture);
  //если кнопка мыши нажата с шифтом, то уменьшить глубину перебо-
ра:
  if (ssLeft in Shift) and (ssShift in Shift) then begin
    if Step> 1 then dec(Step);
    //показать новое значение:
    lblStep.Caption:= inttostr(Step);
    //отцентровать число:
    lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width)
div 2+2;
    exit;
  end;

  //если кнопка мыши нажата без шифта, то увеличить глубину пере-
бора:
  if ssLeft in Shift then begin
    if Step< dgPole.ColCount*dgPole.RowCount then inc(Step);
    lblStep.Caption:= inttostr(Step);
    lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width)
div 2+2;
  end;
end; // imgStepMouseDown
```


Отпуская ее, мы возвращаем на место *серую* кнопку:

```
procedure TForm1.imgStepMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgStep.Picture.Assign(Image11.Picture);
end; // imgStepMouseUp
```

Глубина перебора хранится в глобальной *переменной*

```
Step: integer= 1;
```

Поиск решения активизируется *кнопкой imgSolution*:

```
    Left = 188
    Hint = ' Найти решение '
    OnMouseDown = imgSolutionMouseDown
    OnMouseUp = imgSolutionMouseUp
```

Картинки хранятся в компонентах *Image13* и *Image14* (Рис. 5.24).

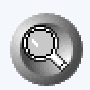
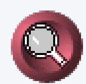
 

Рис. 5.24. «Решительные» кнопки

При нажатии (точнее, при последующем отпускании) кнопки *Solution* программа переводится в режим *gsSolution* и не реагирует на любые команды, кроме остановки поиска:

```
//НАЙТИ РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.imgSolutionMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgSolution.Picture.Assign(Image14.Picture);
end; // imgSolutionMouseDown

procedure TForm1.imgSolutionMouseUp(Sender: TObject; Button:
TMouseButton;  Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgSolution.Picture.Assign(Image13.Picture);
  //перевести программу в режим поиска:
  GameState:= gsSolution;
  //искать решение на заданную глубину:
  solution(step);
  //перевести программу в режим ожидания ввода команд:
  GameState:= gsWait;
end; // imgSolutionMouseUp
```

Самая первая кнопка на форме – *imgNewGame* – запускает новую игру:

```
    Left = 4
    Hint = ' Новая игра '
    OnMouseDown = imgNewGameMouseDown
    OnMouseUp = imgNewGameMouseUp
```

Ее картинки хранятся в компонентах *Image1* и *Image2* (Рис. 5.25).

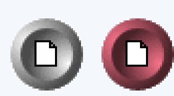


Рис. 5.25. «Запускательные» кнопки

```
//НОВАЯ ИГРА
procedure TForm1.imgNewGameMouseDown(Sender: TObject; Button:
TMouseButton;  Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgNewGame.Picture.Assign(Image2.Picture);
end; // imgNewGameMouseDown

procedure TForm1.imgNewGameMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
```

```
  NewPlay;
  imgNewGame.Picture.Assign(Image1.Picture);
end; // imgNewGameMouseUp
```

Действие этой кнопки заключается в вызове процедуры *NewPlay*, которую мы уже разобрали.

При *ручной* расстановке фишек на поле могут быть очень полезны следующие кнопки.

Вторая справа кнопка – *imgRedOnly* – переворачивает все фишки на игровом поле *красной* стороной вверх:

```
    Left = 44
    Hint = ' Все фишки красные '
    OnMouseDown = imgRedOnlyMouseDown
    OnMouseUp = imgRedOnlyMouseUp
```

Картинки хранятся в компонентах *Image3* и *Image4* (Рис. 5.26).

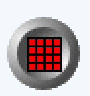


Рис. 5.26. Вгоняем в краску!

```
//ВСЕ ФИШКИ НА ЛЕВОМ ПОЛЕ - КРАСНЫЕ
procedure TForm1.imgRedOnlyMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgRedOnly.Picture.Assign(Image4.Picture);
end; // imgRedOnlyMouseDown

procedure TForm1.imgRedOnlyMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
    if GameState= gsSolution then exit;
    for j:= 0 to dgPole.RowCount-1 do
      for i:= 0 to dgPole.ColCount-1 do
        masPole[i,j]:= BLACK;

    dgPole.Invalidate;
    imgRedOnly.Picture.Assign(Image3.Picture);
end; // imgRedOnlyMouseUp
```

Следующая кнопка – *imgInverse* – изменяет цвет фишек на игровом поле на противоположный:

```
    Left = 84
    Hint = ' Инвертировать '
    OnMouseDown = imgInverseMouseDown
    OnMouseUp = imgInverseMouseUp
```

Картинки хранятся в компонентах *Image5* и *Image6* (Рис. 5.27).

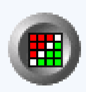
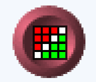
 

Рис. 5.27. «Инверсные» кнопки


```
//ИЗМЕНИТЬ ЦВЕТ ФИШЕК НА ИГРОВОМ ПОЛЕ НА ПРОТИВОПОЛОЖНЫЙ
procedure TForm1.imgInverseMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgInverse.Picture.Assign(Image6.Picture);
end; // imgInverseMouseDown

procedure TForm1.imgInverseMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  if GameState= gsSolution then exit;
  for j:= 0 to dgPole.RowCount-1 do
    for i:= 0 to dgPole.ColCount-1 do
      masPole[i,j]:= BLACK- masPole[i,j];

  dgPole.Invalidate;
  imgInverse.Picture.Assign(Image5.Picture);
end; // imgInverseMouseUp
```

Точно так же действует и кнопка *imgInverse2*, только она меняет цвет фишек на правом поле-образце:

```
    Left = 296
    Hint = ' Инвертировать '
    OnMouseDown = imgInverse2MouseDown
    OnMouseUp = imgInverseMouse2Up
```

Картинки хранятся в тех же компонентах.

```
//ИНВЕРТИРОВАТЬ ЦВЕТ ФИШЕК НА ПРАВОМ ПОЛЕ
procedure TForm1.imgInverse2MouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgInverse2.Picture.Assign(Image6.Picture);
end; //imgInverse2MouseDown
```

```
procedure TForm1.imgInverse2MouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  if GameState= gsSolution then exit;
  for j:= 0 to dgPole2.RowCount-1 do
    for i:= 0 to dgPole2.ColCount-1 do
      masPole2[i,j]:= BLACK- masPole2[i,j];

  dgPole2.Invalidate;
  imgInverse2.Picture.Assign(Image5.Picture);
end; // imgInverse2MouseUp
```


И вот мы добрались до последней кнопки – *imgGreenOnly*. Она родственница кнопки *imgRedOnly*, но переворачивает фишки на правом поле и *зелёной* стороной вверх:

```
Left = 256
Hint = ' Все фишки зелёные  '
OnMouseDown = imgGreenOnlyMouseDown
OnMouseUp = imgGreenOnlyUp
```

Картинки хранятся в компонентах *Image7* и *Image8* (Рис. 5.28).

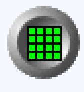
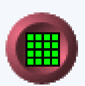
 

Рис. 5.28. Озеленяем!

```
//ПЕРЕВЕРНУТЬ ФИШКИ НА ПРАВОМ ПОЛЕ ЗЕЛЁНОЙ СТОРОНОЙ ВВЕРХ
procedure TForm1.imgGreenOnlyMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgGreenOnly.Picture.Assign(Image8.Picture);
end; // imgGreenOnlyMouseDown

procedure TForm1.imgGreenOnlyMouseUp(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  if GameState= gsSolution then exit;
  for j:= 0 to dgPole2.RowCount-1 do
    for i:= 0 to dgPole2.ColCount-1 do
      masPole2[i,j]:= WHITE;

  dgPole2.Invalidate;
  imgGreenOnly.Picture.Assign(7.Picture);
end; // imgGreenOnlyMouseDown
```

Согласитесь, такие кнопки удобнее размещать на форме, ведь каждая из них содержится только в *одном* компоненте *TImage* (ещё два компонента нужны для хранения картинок, но их можно убрать в любое место на форме и больше не трогать), а не в двух, как в *тетрисе*.

Если вы помните, мы ещё не написали процедуры для двух пунктов меню. *Один* позволяет при решении задач возвращаться на один ход *назад* (вплоть до исходной позиции):

```
//ВЕРНУТЬ ХОД
procedure TForm1.miUndoClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;
  if hod< 1 then exit;

  masPole:= DoMove(masPole, Moves[hod].x, Moves[hod].y);
  dgPole.Invalidate;
  dec(hod);
  lblHod.Caption:=  'Ход - ' + inttostr(hod);
end; // miUndoClick
```


Второй, наоборот, даёт возможность двигаться *вперёд* по уже сделанным, но отменённым ходам:

```
//ОТМЕНИТЬ ПОСЛЕДНИЙ ВОЗВРАТ ХОДА
procedure TForm1.miRedoClick(Sender: TObject);
begin
  if GameState= gsSolution then exit;

  inc(hod);
  masPole:= DoMove(masPole, Moves[hod].x, Moves[hod].y);
  dgPole.Invalidate;
  lblHod.Caption:= 'Ход - ' + inttostr(hod);
end;  // miRedoClick
```

Для большего удобства при решении задач на нечётных досках можно добавить подсказки и для периферийных клеток. Как вы знаете, удачный ход в периферийную клетку уменьшает число нечётных клеток на меньшем поле, а, значит, и общее число ходов, что позволяет найти лучшее решение задачи. Конечно, можно последовательно делать ходы в периферийные клетки и подсчитывать число нечётных клеток, а затем выбирать ту клетку, которая даёт оптимальный результат, но правильнее будет переложить эту рутину на плечи компьютера. Правда, нам придётся поработать головой, но зато потом пользоваться программой будет значительно удобнее.

В первую очередь, нам потребуется *функция*, умеющая подсчитывать количество нечётных клеток на *малом* поле (его ширина и высота на одну клетку меньше игрового поля). Для этого мы проверяем каждую клетку малого поля и, если она нечётная, увеличиваем на 1 возвращаемое значение:

```
//ПОДСЧИТАТЬ ЧИСЛО НЕЧЁТНЫХ КЛЕТОК НА ЗАДАННОМ ПОЛЕ
function TForm1.CalcOdd(Pos: TPole): Integer;
var
  i, j, i1, j1: integer;
  n: integer;
begin
    Result:= 0;
    //размеры малого поля:
    m:= dgPole.ColCount-2;
    //по всем клеткам малого поля:
    for j:= 0 to m do
      for i:= 0 to m do begin
        //число нечётных клеток в соотв. колонке и строке:
        n:= 0;
        for j1:= 0 to m do
          for i1:= 0 to m do
            if (i1= i) or (j1= j) then n:=n+ Pos[i1,j1];

        if odd(n) then inc (Result);
      end;
end; // CalcOdd
```


В процедуре выбора «хороших» периферийных клеток мы сначала определяем, сколько нечётных клеток содержится на малом поле в текущей позиции. Затем последовательно выполняем ход во все периферийные клетки (за исключением угловой) и вновь подсчитываем нечётные клетки. Если их число уменьшилось, то мы запоминаем новое значение. После обхода всех периферийных клеток мы проверяем, удалось ли улучшить результат. Если удалось, отмечаем соответствующие периферийные клетки *кружком* (в отличие от квадрата на малом поле):

```
//НАЙТИ ХОДЫ В ПЕРИФЕРИЙНЫЕ КЛЕТКИ
procedure TForm1.PerifHint(massiv:TPole; Pole: TDrawGrid);
var
  i, j, m, n: integer;
  min0, min: integer;
  Position: TPole;
  Rect: TRect;
begin
  if miHint.Checked= False then exit; //- не выводить подсказки
  //размеры полей:
  m:= Pole.ColCount-1;
```

```
  //подсчитать число нечётных клеток на малом поле в текущей пози-
ции:
  min0:= CalcOdd(massiv);
  min:= min0;

  //проверить периферийные клетки:
  for j:= 0 to m do
    for i:= 0 to m do
      if ((i= m) or (j= m)) and (i<>j) then begin
        //выполнить ход:
        Position:=DoMove(massiv, i, j);
        //подсчитать число нечётных клеток на малом поле:
        n:= CalcOdd(Position);
        //запомнить минимальное число нечётных клеток
        //после хода в периферийную клетку:
        if n< min then min:= n;
       end;

  if min= min0 then exit; //- не удалось уменьшить число ходов

  //отметить периферийные клетки, уменьшающие число ходов:
  for j:= 0 to m do
    for i:= 0 to m do
      if ((i= m) or (j= m)) and (i<>j) then begin
        //выполнить ход:
        Position:=DoMove(massiv, i, j);
        //подсчитать число нечётных клеток на малом поле:
        n:= CalcOdd(Position);
        if n= min then begin
          //нарисовать окружность:
          Rect:= Pole.CellRect(i, j);
          //dgPole.Canvas.Brush.Style:= bsClear;
          //цвет линий:
          Pole.Canvas.Pen.Color:= clBlue;
          Pole.Canvas.Pen.Width:= 1;
          Pole.Canvas.Ellipse(Rect.left+2, Rect.top+2,Rect.right-
2, Rect.bottom-2);
        end;
      end;
end;  // PerifHint
```


Осталось решить, из какого места программы должна вызываться описанная процедура. Удобнее всего это делать в тех процедурах, которые отвечают за прорисовку ячеек игрового и «образцового» полей. Придётся добавить толику кода:

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
```

```
…
  //выводить подсказки для нечётных позиций?
  if (dgPole.ColCount= 5) and ((ACol>3) or (ARow>3)) then
    begin  PerifHint(masPole, dgPole); exit; end;
  if (dgPole.ColCount= 7) and ((ACol>5) or (ARow>5)) then
    begin  PerifHint(masPole, dgPole); exit; end;

…

//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);

…

  //выводить подсказки для нечётных позиций?
  if (dgPole2.ColCount= 5) and ((ACol>3) or (ARow>3)) then
    begin  PerifHint(masPole2, dgPole2); exit; end;
  if (dgPole2.ColCount= 7) and ((ACol>5) or (ARow>5)) then
    begin  PerifHint(masPole2, dgPole2); exit; end;
```


После этих усовершенствований вы легко сможете решить наилучшим образом любую задачу и на нечётных полях 5 x 5 и 7 x 7. Правда, речь до сих пор шла о режиме «полного переворота» на таких полях, но и режим «перехода» вам также доступен. Действуйте на малом поле совершенно по тому же «сценарию», что и на чётных полях. В итоге вы либо сразу совершите переход из одной позиции в другую, либо вам придётся сделать ещё один ход в правый нижний угол, если фишки в этой клетке на игровом и образцовом полях отличаются по цвету.



Мы выяснили, что «нерешаемая» позиция на нечётном поле не может быть сведена к нулевой или любой другой решаемой. Но это ничуть не мешает переводу одной нерешаемой позиции в другую нерешаемую. Наша программа не предлагает для решения такие задачи, но вы можете самостоятельно ввести такие позиции на обоих полях. Не кажется ли вам, что здесь есть над чем подумать?

Например, эти две прекрасные симметричные (но «неправильные») позиции разделяют всего 4 хода (Рис. 5.29):

1. 3 – 1 (3 –2).
2. 1 – 3 (2 –3).
3. 5 – 3 (4 – 3).
4. 3 – 5 (3 – 4).

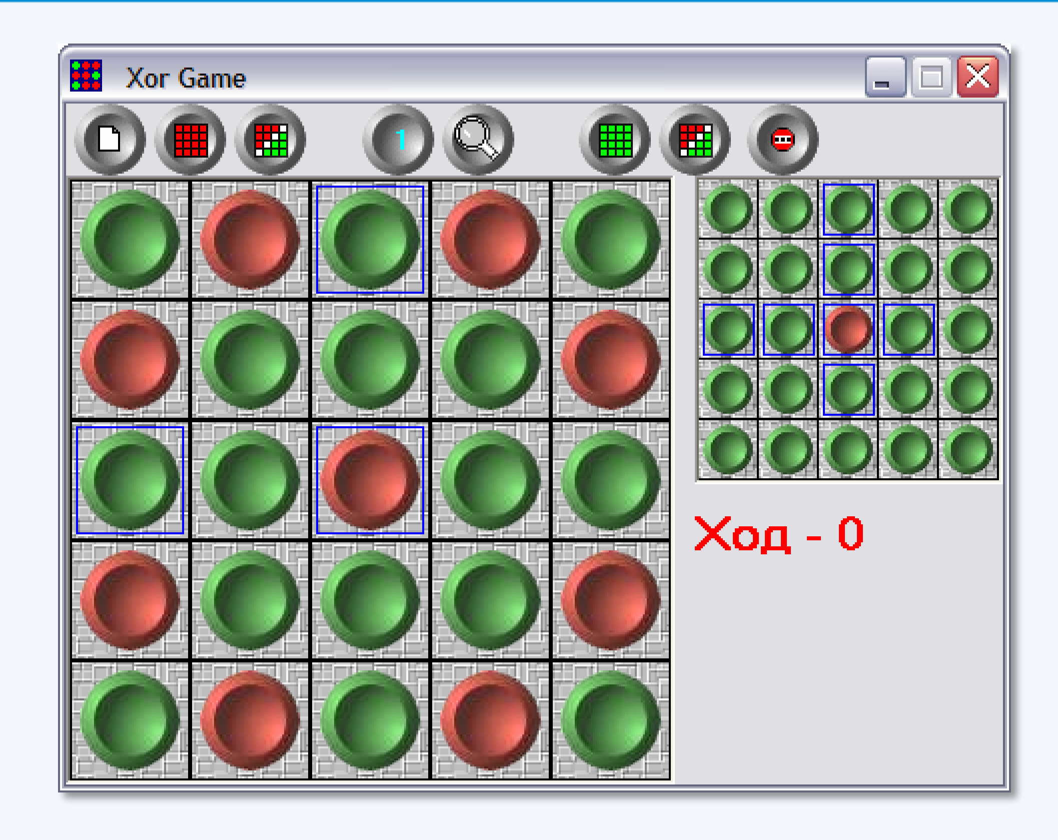


Рис. 5.29. Решаем нерешаемые задачи!

Левая позиция получена из правой методом «блуждания» по правому полю за 9 ходов, но для решения задачи хватит и 4-х, причём самый короткий переход можно найти, пользуясь всё тем же алгоритмом, что и раньше. Таким образом, если между двумя позициями существует переход, то мы сумеем его отыскать. Это значит, что позицию на игровом поле можно задавать произвольно, а позицию на правом поле следует образовывать из позиции на правом с помощью серии случайных ходов. Ясно, что между этими позициями переход всегда возможен, а вот полный переворот фишек на зелёную сторону – не всегда. Если у вас есть желание, вы можете изменить процедуру *NewPlay* так, чтобы в режиме перехода использовались бы и «нерешаемые» позиции. Кстати говоря, среди них вы найдёте множество красивых, симметричных позиций, которые можно использовать в качестве заданий, создав на диске библиотеку избранных задач. Как это сделать, читайте в следующем разделе.

Помоги себе сам, или Даём справочку

Спасение утопающих –
дело рук самих утопающих.

И.Ильф, Е.Петров

Попробуйте доказать, что существуют пары нерешаемых позиций, которые *невозможно* перевести друг в друга. Это трудно, зато я подсоблю вам в написании помощи для этой игры. Вызываться она будет при выборе пункта меню *Правила*. Но в этот раз мы «окажем помощь» более профессионально – с привлечением *Справочной системы Windows*. Такую «справку» вы можете увидеть практически во всех «серьёзных» программах, сделать её нетрудно и самим, особенно если начать с самого простого варианта.

Вывод справки на экран осуществляется в процедуре-обработчике щелчка на пункте меню *Помощь*. Для этого мы используем функцию *Windows API WinHelp*, в которой надлежит указать имя файла справки (у нас это будет *XorGame.hlp*), который должен находиться в той же папке, что и выполняемый файл программы.

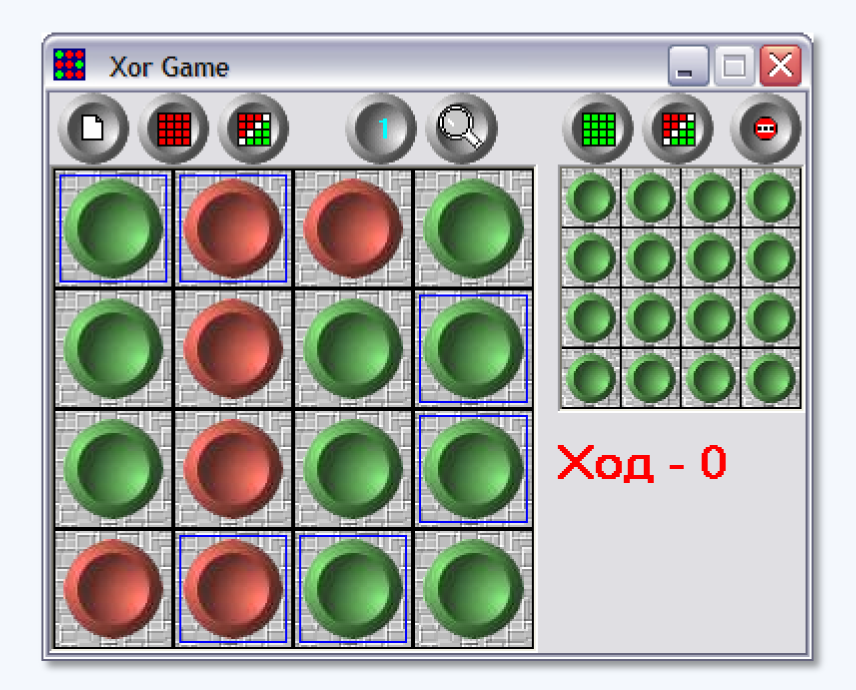
```
//ПОКАЗАТЬ ФАЙЛ СПРАВКИ
procedure TForm1.miHelpClick(Sender: TObject);
begin
  WinHelp(handle, PChar('XorGame.hlp'), HELP_CONTENTS,0)
end;  // miHelpClick
```

Вот только где взять файл с расширением *hlp*? Оказывается, его нетрудно получить из текстового файла в формате *rtf,* который можно создать во многих текстовых редакторах, хотя бы в *WordPad* - он поставляется с *ОС Windows* - или *MS Word*.

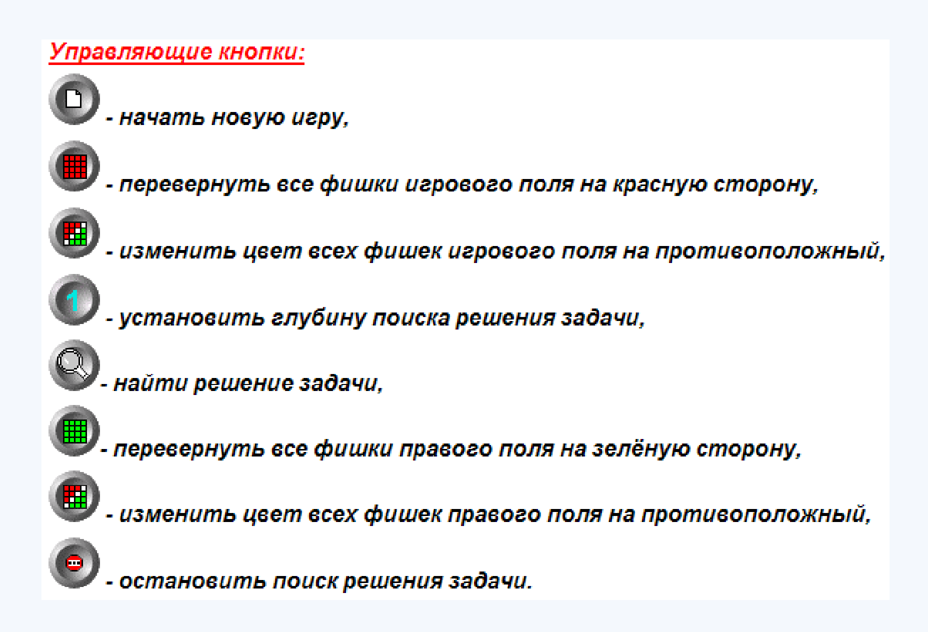
В тексте справки допустимо использовать любые шрифты, а также добавлять картинки. Например, короткая справка для нашей игры могла бы выглядеть так:

XorGame Help

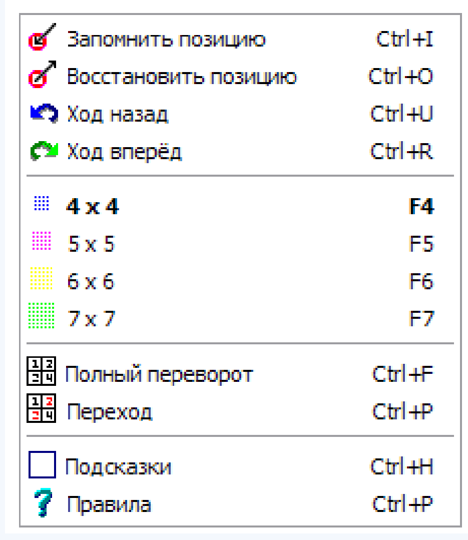
Логическая игра-головоломка, в которой требуется перейти от начальной позиции к конечной за минимальное число ходов, переворачивая на квадратном поле двухцветные фишки.



Ход состоит в переворачивании любой фишки на левом, игровом поле (зелёная фишка становится красной, красная – зелёной), но при этом переворачиваются и все другие фишки в той же колонке и строке. На правом поле вы можете видеть конечную позицию.



Вы также можете управлять программой, вызывая всплывающее меню (для этого нажмите правую кнопку мыши):



Первая группа команд поможет вам найти самое короткое решение задачи. Вы можете запомнить любую позицию на поле, а затем восстановить её, а также вернуться назад на любое число ходов (или вернуть уже сделанные ходы при откате).

Вторая группа команд устанавливает размеры полей. Разумеется, чем больше размер поля, тем труднее будет решить задачу.

У игры имеется две разновидности - полный переворот, когда все фишки на игровом поле нужно перевернуть зелёной стороной вверх, и переход, когда следует перевернуть фишки на игровом поле так, как они расположены на правом поле-образце.

Если включен режим подсказки, то программа будет отмечать те клетки игрового поля, на которые нужно ходить.

Закончив набор, сохраните файл в формате *RTF* (в этом случае имя файла справки на диске будет *XorGame.rtf*).

Затем запустите программу *Microsoft Help Workshop* под именем *hcw.exe* (в *Turbo Delphi* она отсутствует, но в папке *Tools* вы найдёте её копию, взятую из *Delphi 5*). Выполните команду *File|New…* и в открывшемся диалоговом окне нажмите кнопку *OK* (по умолчанию будет создаваться *Help Project*). В следующем диалоговом окне перейдите в папку с программой и введите имя файла справки (без расширения). Снова нажмите кнопку *OK*, и вы окажетесь в окне с вашим проектом. *Help Workshop* самостоятельно запишет в него строки

```
[OPTIONS]
LCID=0x419 0x0 0x0 ; Русский
REPORT=Yes
```

Сохраните файл на диске (к заданному вами имени файла добавится расширение *hpj*), а потом откройте в любом текстовом редакторе (лучше всего подойдёт *Блокнот* из *ОС Windows*) и допишите следующие строки:

```
TITLE=XorGame Help
COPYRIGHT=В.Рубанцев, 2007
HLP=.\XorGame.hlp
[FILES]
XorGame.RTF
```

TITLE – это название справки, которое появится на панели задач при запуске помощи.

COPYRIGHT – фамилия автора программы, она выводится при выборе пункта меню *Справка|Версия* в окне справочной системы *Windows* с вашей справкой.

HLP – размещение файла помощи и его имя.

В разделе *[FILES]* следует указать имя файла справки с расширением *RTF*.

То же самое можно сделать и в самой программе *Help Workshop*, нажав кнопку *Options* (Рис. 5.30).

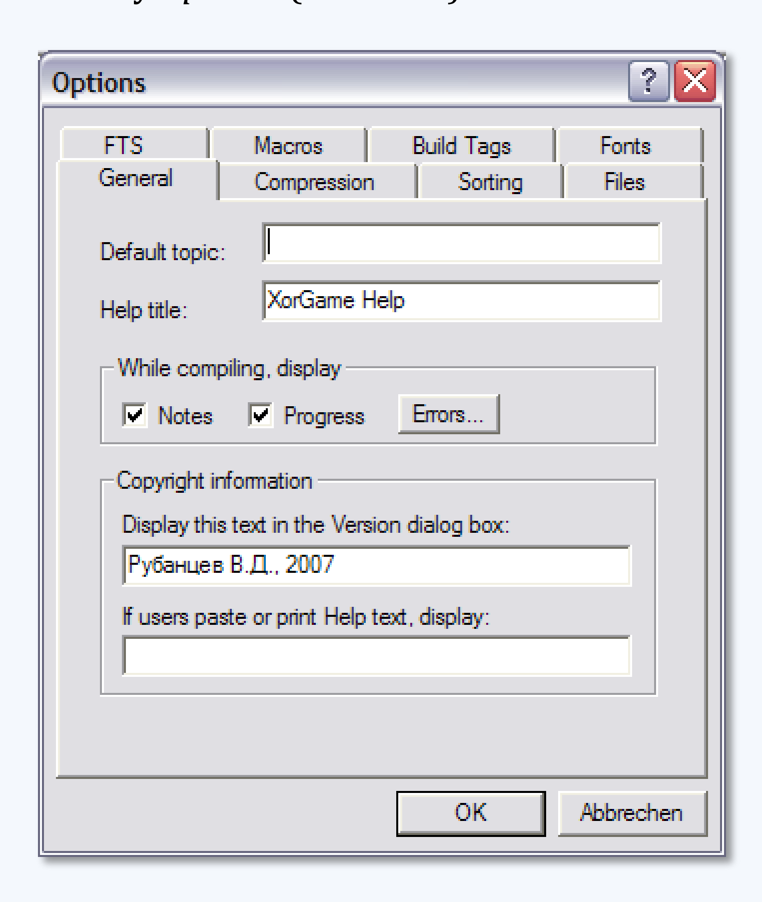


Рис. 5.30. Создаем справочный файл

Сохраните отредактированный файл проекта. Осталось только скомпилировать его в файл помощи. Запустите *Microsoft Help Workshop* (или дважды щёлкните на вашем файле с расширением *hpj* на диске). В открывшемся окне вы увидите название файла помощи и его текст. Нажмите кнопку *Save and Compile* (она находится в правом нижнем углу формы), и через мгновение на диске появится файл помощи с расширением *hlp*. Щёлкните на нём дважды и убедитесь, что справочный файл выглядит именно так, как вы и планировали. Если необходимо, внесите изменения и дополнения в файл справки и проекта, а после этого перекомпилируйте проект с помощью *Microsoft Help Workshop*.

Точно такое же окно с вашей справкой будет появляться, когда пользователь выберет пункт *Правила* игры *XorGame*. Вероятно, такой способ создания справки к программам вам покажется более привлекательным, чем рассмотренные нами ранее, - сил и времени набор текста справки отнимает немного, а результаты дает впечатляющие! Тем более что вы можете добавить к справке и другую полезную информацию. Сделать это уже сложнее, поэтому мы пока ограничимся только простейшими возможностями *Справочной системы Windows*.



Операционная система *Windows 7* не поддерживает этот формат справочных файлов. Имейте это в виду!



Исходный код программы находится в папке **xor**.

Есть варианты, или ShortGame

Кроме игры *Флип-Флоп*, есть ещё несколько подобных игр, из которых мы подробно остановимся только на одной, а короткому обзору остальных посвятим следующий раздел.

Основной признак всех игр-оборотней – это переворот двухцветных фишек на квадратном поле (в игре *Камешки* поле не квадратное, а прямоугольное, но вы можете сделать его и крестообразным и гексагональным…), поэтому близкие родственники игры *Флип-Флоп* отличаются от неё только правилом (порядком, шаблоном) переворачивания фишек.



Если нарушить это условие и взять фишки с тремя или четырьмя сторонами, окрашенными в два или более цветов, то можно создать очень любопытные разновидности игр-оборотней. Если пойти ещё дальше и *выставлять* на поле по одной фишке, то можно получить прекрасную игру *Logos*, о которой будет рассказано в четвертом факультативе.

Так, в игре *Blinkin'* (версия 1.0, 1999 г.) (Рис. 5.31), которая послужит нам основой для очередной игры, фишки переворачиваются точно так же, как и в *XorGame*, но только на расстояние в 1 фишку во все стороны от «центральной». Поэтому вполне естественно назвать нашу программу *коротким Xor'ом*, или *ShortGame*.



На рисунке ясно видно, что «нормальный» ход возможен только в *середине* поля, а у *границ* и в *углах* шаблон очень сильно искажается. Но вы легко сможете уберечь шаблон от «ампутации», если возьмёте тороидальное поле (в виде бублика). На экране оно так и останется квадратным (прямоугольным), но первый и последний ряды (столбцы и колонки) нужно считать *соседними*, то есть «лишние» фишки должны появляться на *другой* стороне поля. Запрограммировать эти особенности игры на торе не составит большого труда.

Вы также можете попытаться изменить длину «лучей» у стандартного шаблона или сделать их диагональными. А вдруг вам удастся придумать новые игры-оборотни!

Для преобразования достаточно исправить процедуры выполнения хода, а всё остальное можно оставить практически без изменений. Но этот путь

слишком лёгкий для нас, а, значит, и не очень интересный и не очень полезный, поэтому мы дополнительно изменим дизайн программы и добавим ещё несколько симпатичных новинок.

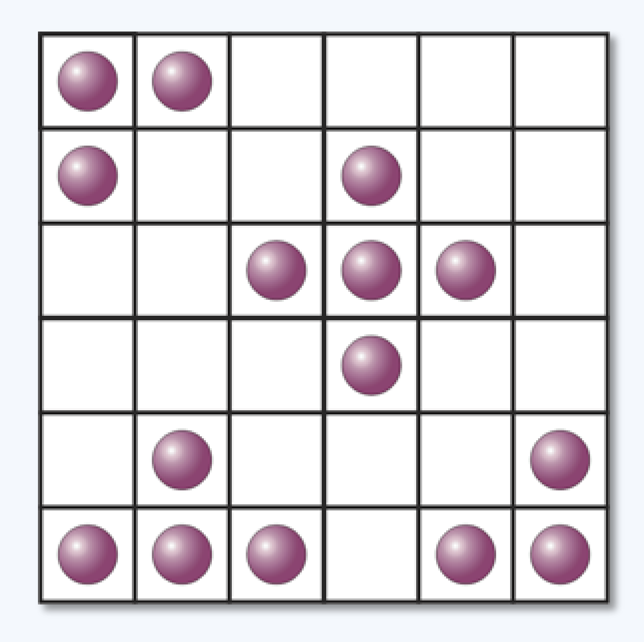


Рис. 5.31. Варианты ходов (шаблоны для переворачивания фишек) в игре *Blinkin'*

Зачем нужны фиксированные ряды?

При изучении свойств компонента *TDrawGrid*, я отмечал, что *фиксированные* ряды могут использоваться для *оцифровки* игровых полей. В этой игре я покажу вам два способа оцифровки. В *первом* все числа и буквы будут изображены на растровых картинках, во *втором* – просто напечатаны стандартным шрифтом. Вы можете оценить достоинства и недостатки обоих способов и выбрать подходящий. Хотя если вы можете нарисовать в графическом редакторе красивые картинки, то второй способ вам никогда и не понадобится – результаты его применения далеки от требований сегодняшнего дня.

Чтобы не переписывать весь код приложения заново, откройте в *Turbo Delphi* проект *XorGame* и сохраните все модули, а затем и сам проект под новыми именами (кроме формы протокола). Теперь вы можете исправлять и добавлять новый код в проект игры *ShortGame* (Рис. 5.32).

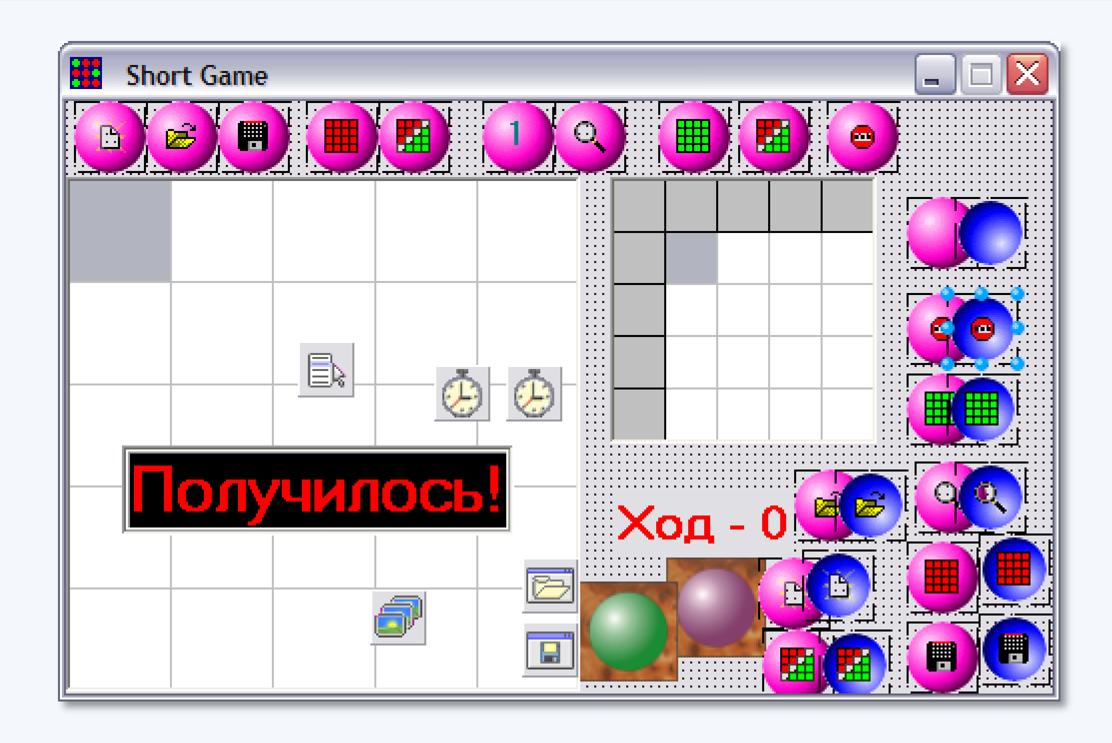


Рис. 5.32. Главная форма проекта в окне *Конструктора формы*

Изменения начнём с *сеток*. Свойства сетки *dgPole* (указаны только те свойства, которые нужно исправить):

```
    Top = 38
    Width = 257
    Height = 257  (немного опустили сетку, чтобы кнопки чувствова-
ли себя более уютно, размеры сетки увеличились из-за того, что по-
требовались ещё ряды для оцифровки)
    ColCount = 5
    RowCount = 5
    DefaultColWidth = 50
    DefaultRowHeight = 50  (размеры клеток пришлось уменьшить, так
как в игре будут использоваться поля до 9 x 9 клеток)
    GridLineWidth = 1  (очертим все клетки – фон клеток стал тем-
нее, и чёрные линии будут плохо заметны)
   Options = [goVertLine, goHorzLine]
```

Свойства сетки *dgPole2*:

```
    Left = 272
    Top = 38
    Width = 133
    Height = 133
    Cursor = crHandPoint
    ColCount = 5
    RowCount = 5
    DefaultColWidth = 25
    DefaultRowHeight = 25
    FixedColor = clSilver
```

```
    FixedCols = 1
    FixedRows = 1
    GridLineWidth = 1
    Options = [goFixedVert-
Line,goFixedHorzLine,goVertLine,goHorzLine]
```

На *правой* сетке мы разместим два фиксированных ряда – горизонтальный и вертикальный – для вывода оцифровки. Это сделано только для примера – никаких преимуществ этот способ оцифровки не даёт.

Перейдём к процедурам рисования клеток игрового поля. Ради разнообразия мы заменим фишки шариками на «паркете» (Рис. 5.33).



Рис. 5.33. Шариковые фишки

Для оцифровки потребуется довольно много картинок (Рис. 5.34).

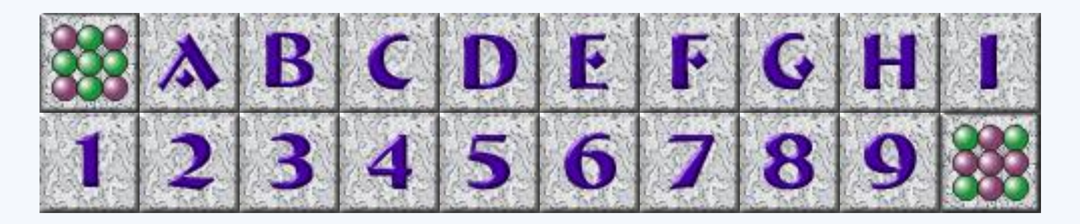


Рис. 5.34. Картинки для оцифровки полей

Как и раньше, их можно загрузить в компоненты *TImage*, расположенные на форме, но картинок так много, что разумнее использовать для их хранения специальный компонент *TImageList* (страница *Win32* в *Палитре компонентов*). Впрочем, это создаёт и определённые неудобства, в чём вы скоро сами сможете убедиться.

Итак, щёлкните на этом компоненте в *Палитре*, и он послушно возникнет на форме. Сразу же установите размеры картинок 50 x 50 пикселей, иначе при загрузке они будут искажены:

```
    Name = imlNota
    Height = 50
    Width = 50
```

А вот теперь откройте *Редактор*, дважды щёлкнув компонент *TImageList* (тот, который на форме, естественно) (Рис. 5.35).

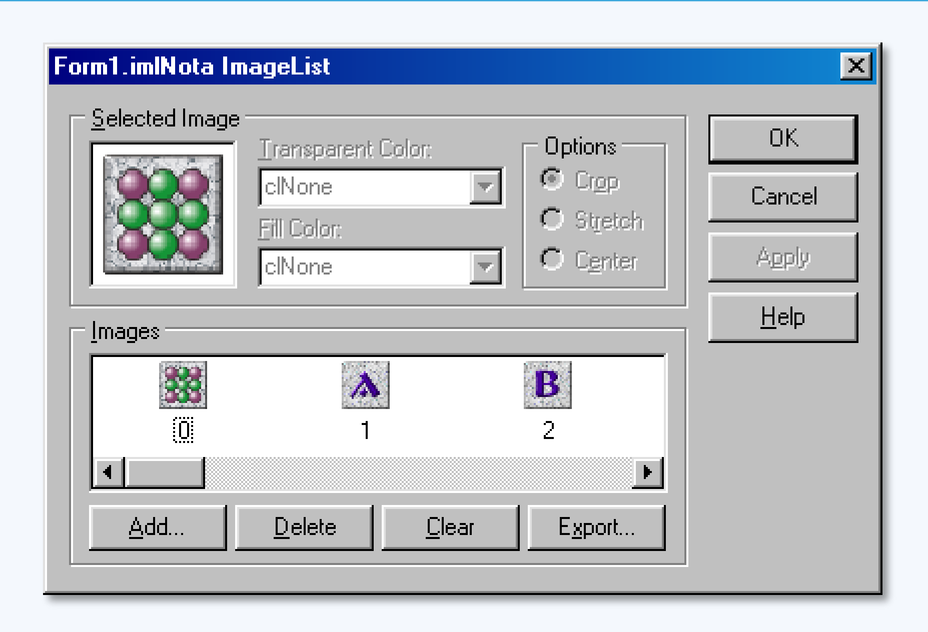


Рис. 5.35. Картинки в списке!

Нажимайте кнопку *Add…* и загружайте заранее подготовленные картинки с оцифровкой. Их примерный облик представлен выше. Порядок загрузки картинок – тоже. Под каждой картинкой вы увидите число – это *индекс* соответствующей картинки в списке, он используется для ее вывода на канву компонента. Как только список заполнится, закрывайте *Редактор* и переходите к написанию кода.

В процедуре *dgPoleDrawCell* вычисляется «номер» *m* клетки с координатами *ACol, ARow* на игровом поле, а по нему определяется дальнейшая судьба клетки. В *верхнюю* строку выводятся картинки с индексами 0-9 (*буквы*), а в *первую* колонку - картинки с индексами 10-18 (*цифры*), во все остальные – шарики соответствующего цвета.

Чтобы перенести картинку из компонента *TImageList* на канву сетки, приходится использовать его метод

```
procedure Draw(Canvas: TCanvas; X, Y, Index: Integer; Enabled:
Boolean=True);
```

Из чего следует, что вы не сможете непосредственно применить более мощные методы самой канвы.

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var
  n, m: integer;
  r, dr: TRECT;
```

```
begin
  //фишка в ячейке:
  n:= masPole[ACol, ARow];
  //размеры картинок:
  r:= Bounds(0, 0, 50, 50);
  dr:= Bounds(Rect.Left, Rect.Top, 50, 50);
  m:= 100 * ARow + ACol;
  case m of
    0..9: //- верхняя строка
      imlNota.Draw(dgPole.Canvas, dr.left, dr.top, m);
    100, 200, 300, 400, 500, 600, 700, 800, 900: //- первая
колонка
      imlNota.Draw(dgPole.Canvas, dr.left, dr.top, m div 100 + 9);
    else begin
      case n of
        WHITE:    //- белая фишка
          dgPole.canvas.CopyRect(dR, imgWhite.Canvas, R);
        BLACK:    //- чёрная фишка
          dgPole.canvas.CopyRect(dR, imgBlack.Canvas, R);
      end
    end
  end;
end;  //dgPoleDrawCell
```


В остальном эта процедура не таит в себе никаких неприятных сюрпризов. Обратите внимание на то, что подсказки теперь не выводятся. Правила выбора ходов для игры *XorGame* здесь работать не будут, поэтому их нужно заменить другими. А вот какими – попробуйте найти сами!

В правой сетке имеются фиксированные ряды для печати оцифровки, но – на всякий случай – предусмотрена и возможность вывода тех же картинок с буквами и цифрами, что и на игровом поле, но в масштабе 1 : 2.

Для букв и цифр выбран *жёлтый* цвет на сером фоне, но вы легко можете заменить их другими. Вывод символов осуществляется методом канвы

```
procedure TextRect(Rect: TRect; X, Y: Integer; const Text:
string);
```

Сделать это чрезвычайно просто, и если вас удовлетворяет результат, то вы можете смело пропустить следующий абзац.

Вывод картинок на канву *правой* сетки осложняется тем, что размер картинок в компоненте *ImageList1* отличается от размеров клеток на правом поле. Конечно, можно заготовить ещё один комплект букв и цифр подходящего размера, но это не лучший вариант. Если бы картинки хранились в компоненте *TImage*, то всё бы решалось так же просто, как и для картинок с шариками. Однако для картинок с оцифровкой недоступен

метод канвы *StretchDraw*. Вместо него мы должны применять метод *Draw* компонента *TImageList*, который не позволяет масштабировать изображения. Придётся создать в памяти компьютера растр *bmp* по размеру картинок, скопировать в него изображение из компонента *TImageList* и только потом перенести в клетку в уменьшенном масштабе. Чтобы воспользоваться этим способом, раскомментируйте строчки, в которых присутствует *bmp* и закомментируйте вывод текста.

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);
var
  n, m: integer;
  r, dr: TRECT;
  //bmp: TBitmap;
  ch: string;
begin
  //фишка в ячейке:
  n:= masPole2[ACol, ARow];
  //размеры картинок:
  r:= Bounds(0, 0, 25, 25);
  dr:= Bounds(Rect.Left, Rect.Top, Rect.Right-
Rect.Left,Rect.Bottom-Rect.Top);

  m:= 100 * ARow + ACol;

  //создать промежуточный растр:
  //bmp:= TBitmap.Create;
  //bmp.Width:= 50;
  //bmp.Height:= 50;


  //оцифровка:
  dgPole2.Canvas.Font.Color:= clYellow;
  dgPole2.Canvas.Font.Name:='MS Sans Serif';
  dgPole2.Canvas.Font.Style:= [fsBold];
  dgPole2.Canvas.Font.Size:= -14;
  dgPole2.Canvas.Brush.Color:= clGray;

  case m of
    0: //- угловая клетка
      begin
        imlNota.Draw(bmp.Canvas, 0, 0, 0);
        dgPole2.canvas.StretchDraw(dR, bmp);
      end;

    1..9: //- верхняя строка
      begin
        //imlNota.Draw(bmp.Canvas, 0, 0, m);
        //dgPole2.canvas.StretchDraw(dR, bmp);
```

```
        ch:= letter[m];
        with dR, dgPole2.Canvas do
          textrect(dr, left+(right-left-textwidth(ch)) div 2,
             top+(bottom-top-textheight(ch)) div 2, ch);
      end;

    100, 200, 300, 400, 500, 600, 700, 800, 900: //- первая
колонка
      begin
        //imlNota.Draw(bmp.Canvas, 0, 0, m div 100 + 9);
        //dgPole2.canvas.StretchDraw(dR, bmp);
        ch:= inttostr(m div 100);
        with  dR,dgPole2.Canvas do
          textrect(dr, left+(right-left-textwidth(ch)) div 2,
top+(bottom-top-textheight(ch)) div 2, ch);
      end;
    else begin
      case n of
        WHITE:    //- белая фишка
          dgPole2.canvas.StretchDraw(dR,
imgWhite.Picture.Graphic);
        BLACK:   //- чёрная фишка
          dgPole2.canvas.StretchDraw(dR,
imgBlack.Picture.Graphic);
      end
    end
  end;
  //bmp.Free;
end;  //dgPole2DrawCell
```


Если вы помните, у нас остались ещё две картинки в компоненте *ImageList1* – первая и последняя. Они предназначены для заполнения пустой клетки в левом верхнем углу сетки, в которой отсутствуют символы оцифровки. Можно ограничиться только первой картинкой, которая выводится в процедуре *dgPoleDrawCell*, но мы сделаем простейшую *анимацию*. Всякое оживление изображения заключается в последовательной смене картинок на экране. У нас таких картинок всего две, и мы будем поочерёдно выводить их в угловую клетку. Промежуток времени между сменой изображений проще всего отмерять *таймером*. Один таймер у нас имеется, но он уже занят, поэтому добавим к проекту второй таймер *Timer2*. Интервал установите в 500 мс (полсекунды). Чтобы следить за тем, какая в данный момент картинка на экране, мы заведём глобальную *переменную*

```
flgFlash: boolean= false;
```

При каждом срабатывании таймера мы будем изменять её значение на противоположное, что позволит нам определить индекс картинки для вывода.

Дважды щёлкните на значке компонента *Timer2* и в заготовке процедуры напишите код:

```
//МИГАЮЩИЙ КВАДРАТИК
procedure TForm1.Timer2Timer(Sender: TObject);
var
  rect: TRect;
  n: integer;
begin
  rect:= dgPole.CellRect(0,0);
  flgFlash:= not flgFlash;
  if flgFlash= false then n:= 0 else n:= 19;
  imlNota.Draw(dgPole.Canvas, rect.left, rect.top, n);
end;  //Timer2Timer
```

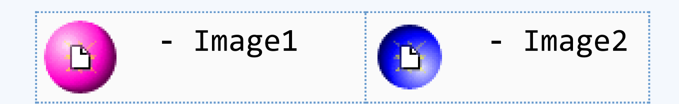



Немного усложнив эту процедуру, вы сможете сделать и более качественную анимацию. Смотрите пример в главе, посвящённой игре *Logos*.

Кнопки

В этой игре мы сделаем кнопки по тому же принципу, что и раньше, но при нажатии они будут изменять не только цвет, но и *размеры*. Изготовить их можно в растровом графическом редакторе и записать в формате *BMP* с одноцветным фоном. Хранить их придётся на форме в компонентах *TImage* (но можно и «по-новому» - в компоненте *TListImage*).

Итак, загружаем картинки с кнопками в соответствующие компоненты (их свойства те же, что и в предыдущей игре) (Рис. 5.36).



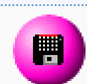
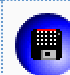
Сразу установим свойства компонентов *Image* для «настоящих» кнопок, которые находятся над сетками:

```
Name = imgNewGame
Left = 4
Top = 0  (такое же значение и у всех других кнопок)
Width = 36
Height = 36
AutoSize = True
Cursor = crHandPoint
Hint = 'Новая игра'
ShowHint = True
```

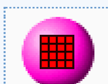
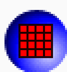
```
Transparent = True
OnMouseDown = imgNewGameMouseDown
OnMouseUp = imgNewGameMouseUp
```

 - Image9	 - Image10

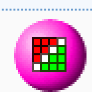
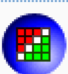
```
Name = imgOpen
Left = 40
Hint = ' Загрузить задачу с диска '
OnMouseDown = imgOpenMouseDown
OnMouseUp = imgOpenMouseUp
```


 - Image18	 - Image19

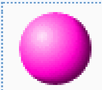
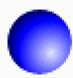
```
Name = imgSavePos
Left = 76
Hint = ' Записать текущую позицию '
OnMouseDown = imgSavePosMouseDown
OnMouseUp = imgSavePosMouseUp
```

 - Image3	 - Image4

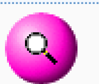
```
Name = imgRedOnly
Left = 120
```

 - Image5	 - Image6

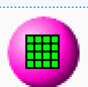
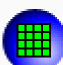
```
Name = imgInverse
Left = 156
```

 - Image11	 - Image12

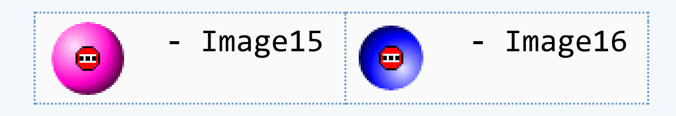
```
Name = imgStep
Left = 208
```

 - Image13	 - Image14

```
Name = imgSolution
Left = 244
```

 - Image7	 - Image8

Name = imgGreenOnly
Left = 296



Name = imgStop
Left = 380

Рис. 5.36. Новые кнопки!

Теперь наполним глубоким смыслом и содержанием *процедуры*, которые будут вызываться при нажатии кнопок. Некоторые из них останутся в неизменном виде, поэтому их код здесь не приводится.

Минимальным изменениям подверглась процедура для кнопки *imgStep*. Они касаются только центрирования надписи с числом шагов:

```
//ЗАДАТЬ ГЛУБИНУ ПЕРЕБОРА
procedure TForm1.imgStepMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
Begin
…
    //отцентровать число:
    lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width)
div 2-1;
…
    lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width)
div 2-1;
    lblStep.Top:= imgStep.Top+ (imgStep.Height- lblStep.Height)
div 2-1;
…
end; //imgStepMouseDown

procedure TForm1.imgStepMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
…
  lblStep.Left:= imgStep.Left+ (imgStep.Width- lblStep.Width) div
2;
  lblStep.Top:= imgStep.Top+ (imgStep.Height- lblStep.Height) div
2;
end; //imgStepMouseUp
```

При срабатывании кнопки *imgRedOnly* позиция на поле изменяется, поэтому её следует запомнить:

```
procedure TForm1.imgRedOnlyMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
…
    dgPole.Invalidate;
    MemoryIn;
  imgRedOnly.Picture.Assign(Image3.Picture);
end; //imgRedOnlyMouseUp
```

Эту же строку добавьте в процедуры *imgGreenOnlyMouseUp*, *imgInverseMouseUp* и *imgInverse2MouseUp*.

В этой игре добавились ещё две кнопки. *Одна* позволяет сохранить позицию на игровом поле:

```
//ЗАПИСАТЬ ПОЗИЦИЮ НА ДИСК
procedure TForm1.imgSavePosMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSavePos.Picture.Assign(Image19.Picture);
end; //imgSavePosMouseDown

procedure TForm1.imgSavePosMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSavePos.Picture.Assign(Image18.Picture);
  SavePos;
end; //imgSavePosMouseUp
```

Вторая – загрузить позицию с диска:

```
//ЗАГРУЗИТЬ ЗАДАЧУ
procedure TForm1.imgOpenMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgOpen.Picture.Assign(Image9.Picture);
  Open;
end; //imgOpenMouseUp

procedure TForm1.imgOpenMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  if GameState= gsSolution then exit;
  imgOpen.Picture.Assign(Image10.Picture);
  //Open;
end; //imgOpenMouseUp
```

Для их работы потребуются компоненты *TSaveDialog* TSaveDialog и *TOpenDialog* TOpenDialog (страница *Dialogs Палитры компонентов*).

Чтобы записать позицию на игровом поле, достаточно скопировать массив поля *masPole* в файл на диске. Обратите внимание, что для хранения задач (позиций) нужно завести *папку Figures* в той папке, в которой находится выполняемый файл программы. Второе замечание: название задачи (имя файла), для удобства, выводится в заголовке формы. Так делается во многих программах, и мы не будем отступать от этой доброй традиции. Но придётся объявить *константу* с названием программы:


```
NAME_PROG =' ShortGame';
```

и *переменную* – для хранения имени файла:

```
NameFig: string= '';
```

И мне осталось напомнить вам, что процедура *SavePos* также должна быть объявлена – в описании типа формы:

```
  private
    { Private declarations }
    procedure Prepare(Pole: TDrawGrid; ColCount, RowCount: inte-
ger);
    function IsReady: Boolean;
    procedure NewPlay;
    procedure Solution(nStep: integer);
    procedure Ready;
    function DoMove(arr: TPole; x, y: integer): TPole;
    procedure MemoryIn;
    procedure SavePos;

//ЗАПИСАТЬ ПОЗИЦИЮ НА ИГРОВОМ ПОЛЕ
procedure TForm1.SavePos;
var
  F: textfile;
  fn,s: string;
  i,j: integer;
  ch: char;
begin
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Задачи (*.txt)|*.TXT';
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'Figures\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите позицию на диск';
  if NameFig<>'' then savedialog1.filename:= NameFig
```

```
  else savedialog1.filename:='temp.txt';
  if not savedialog1.Execute then exit;

  //имя конечного файла:
  fn:= savedialog1.filename;
  NameFig:=fn;
  assignfile(f,fn);
  rewrite(f);

  //записать позицию:
  for j:= 1 to dgPole.RowCount-1 do begin
    s:='';
    for i:= 1 to dgPole.ColCount-1 do begin
      if masPole[i][j]= 1 then ch:='1'
      else ch:='0';
      s:=s+ch;
    end;
   writeln (f,s);
  end;

  closefile(f);
  form1.caption:= NAME_PROG + '   [' + NameFig + ']';
  messagebeep(0)
end;  //SavePos;
```


Обратное действие, то есть загрузка задачи с диска, ненамного сложнее: дополнительно придётся *проверить* данные задачи (файл с условием задачи - обычный текстовый файл, поэтому его можно набрать в любом текстовом редакторе; это удобно, но может привести к ошибкам) и настроить размеры полей, ведь они могут измениться. Труднее всего выделить пункт в меню, соответствующий размерам нового поля, но и с этой проблемой можно справиться.

```
//ЗАГРУЗИТЬ ЗАДАЧУ
procedure TForm1.Open;
var
  s: string;
  F: TextFile;
  nLines,Len: integer;
  i,j:integer;
  ss: array[1..MAX_POLE_HEIGHT] of string;
  comp: TComponent;
begin
  form1.opendialog1.DefaultExt:='txt';
  form1.opendialog1.Filter:='Text files (*.txt)|*.TXT';
  s:=extractfilepath(application.exename)+'FIGURES\';
  form1.opendialog1.InitialDir:= s;
  form1.opendialog1.Title:='Загрузите новую задачу';
  if not form1.opendialog1.Execute then exit;
```

```
  s:= form1.opendialog1.filename;
  //название задачи:
  NameFig:=s;
  {$i-}
  AssignFile(F,NameFig);
  Reset(F);
  {$i+}

  if IOResult<>0 then begin  //- ошибка при загрузке файла
    application.MessageBox ('Такой сетки нет!',NAME_PROG, MB_OK);
    exit
  end;

  nLines:=0;
  Len:=0;
  while not eof(f) do begin
    inc(nLines);
    //считать строку из файла:
    Readln(F, S);
    ss[nLines]:=s;
    If (Length(s) <> Len) and (Len<>0) Then begin
      application.MessageBox ('Неверная длина строки!',NAME_PROG,
MB_OK);
      exit
    end
    else
      Len:= Length(s);
  end;

  //закрыть файл:
  CloseFile(F);

  //проверить данные:
  if (Len> MAX_POLE_WIDTH) or (nLines> MAX_POLE_HEIGHT) then begin
    application.MessageBox('Неверные данные!',NAME_PROG, MB_OK);
    exit
  end;
  form1.caption:= NAME_PROG + '    [' + NameFig + ']';

  //вывести новые поля:
  Prepare(dgPole, Len+1, nLines+1);
  Prepare(dgPole2, Len+1, nLines+1);

  //очистить массивы полей:
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do begin
      masPole[i,j]:= WHITE;
      masPole2[i,j]:= WHITE
    end;

  //заполнить массив новыми данными:
  for j:=1 to nLines do begin
```

```
    for i:=1 to Len do begin
      if ss[j,i]= '0' then
        masPole[i][j]:= 0
      else
        masPole[i][j]:= 1;
    end
  end;

  //запомнить начальную позицию:
  MemoryIn;
  dgPole.Invalidate;
  dgPole2.Invalidate;

  //выделить пункт в меню, соотв. размеру поля:
  comp:=FindComponent('mi'+inttostr(len)+'x'+inttostr(len));
 (comp as TMenuItem).Checked:= True;
end; //Open;
```


Пользуясь этими процедурами, вы можете создать *библиотеку заданий* на диске и в будущем использовать свои лучшие находки. Дальнейшее развитие этой темы вы найдёте в описании игры *Logos*.

Будет разумно сохранять на диске и *решения* задач, что и сделано в модуле формы *frmProtokol* (Рис. 5.37).

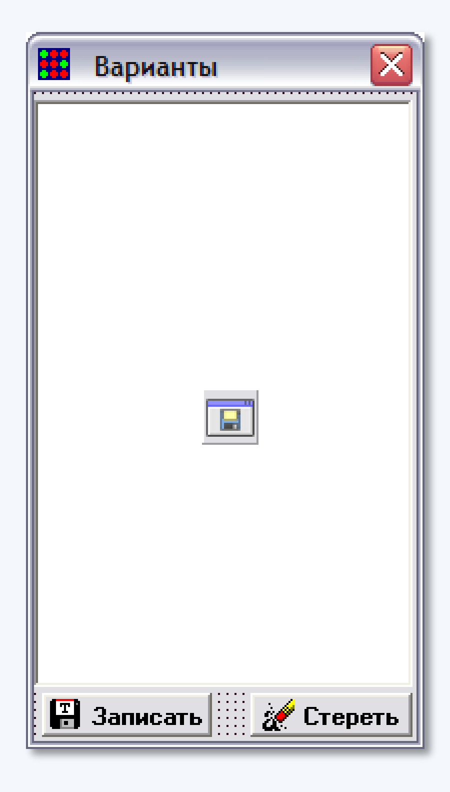


Рис. 5.37. Протокольная форма!

```
unit ProtokolUnit;
```

```
interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, Buttons;

type
  TfrmProtokol = class(TForm)
    ListBox1: TListBox;
    sbtSaveOtv: TSpeedButton;
    SpeedButton1: TSpeedButton;
    SaveDialog1: TSaveDialog;
    procedure SpeedButton1Click(Sender: TObject);
    procedure sbtSaveOtvClick(Sender: TObject);

  privaterivate
    { Private declarations }

  public
    { Public declarations }
  end;

var
  frmProtokol: TfrmProtokol;

implementation

uses ShortGame;

{$R *.DFM}

//ОЧИСТИТЬ СПИСОК
procedure TfrmProtokol.SpeedButton1Click(Sender: TObject);
begin
  ListBox1.Items.Clear;
end;

//ЗАПИСАТЬ ОТВЕТ
procedure TfrmProtokol.sbtSaveOtvClick(Sender: TObject);
var
  s,ss,fn: string;
  i: integer;
  f: textfile;
begin
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Решения (*.txt)|*.TXT';
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'Figures\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите решение на диск';
```

```
  if NameFig<>'' then begin
    s:=extractFileName(NameFig);
    //listbox1.items.add (s);
    ss:='';
    i:=1;
    while (s[i]<>'.') and (i<=length(s)) do begin
      ss:=ss+s[i];
      inc(i)
    end;
    ss:=ss+'otv.txt';
    //listbox1.items.add (ss);
    savedialog1.filename:=ss
   end
   else
     savedialog1.filename:='otv_temp.txt';

   if not savedialog1.Execute then exit;

   //имя конечного файла:
   fn:= savedialog1.filename;
   assignfile(f,fn);
   rewrite(f);
   for i:=0 to (ListBox1.Items.Count - 1) do
     writeln (f,ListBox1.Items.Strings[i]);
   Closefile(f);
   Messagebeep(0)
 end;
end.
```


Теперь пройдёмся по остальным процедурам программы *ShortGame*.

При создании формы нужно увеличить размеры полей до 5 x 5, так как вокруг них добавилась оцифровка:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //подстраиваем величину формы под размеры сеток:
  AutoSize:= True;
  //установить размеры полей по умолчанию:
  Prepare(dgPole, 5, 5);
  Prepare(dgPole2, 5, 5);
  //начать новую игру:
  NewPlay;
end; //FormCreate
```

Основные изменения в процедуре *NewPlay* связаны с выводом названия задачи в заголовок формы и ограничением числа ходов в процедуре возврата ходов *Redo*. В глобальной *переменной*

```
  Hod_max: integer= 0;
```

будет храниться число ходов, сделанных при решении задачи, тогда процедура *Redo* не уйдёт «за границу».

В *ShortGame* даже на поле 4 x 4 клетки возможны *нерешаемые* задачи, поэтому я оставил прежний алгоритм для создания случайной позиции на полях. Впрочем, никакая другая позиция, кроме нулевой, на поле-образце в данной программе не используется и оно присутствует только для дальнейшего развития игры.


```
//ПОДГОТОВКА К НОВОЙ ИГРЕ
procedure TForm1.NewPlay;
var
 i, j: integer;
 x, y: integer;
begin
  //обнулить число ходов:
  Hod:= 0; Hod_max:= 0;
  lblHod.Caption:=  'Ход - 0';

  //задача не имеет названия:
  form1.caption:= NAME_PROG + '   []';

  //сделать случайные ходы:
  randomize;
  //очистить левое поле:
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do
      masPole[i,j]:= WHITE;
  //сделать случайные ходы:
  for i:= 1 to 7 + random(dgPole.ColCount*dgPole.RowCount+1-7) do
begin
    x:= Random(dgPole.ColCount-1)+1; y:= Random(dgPole.RowCount-
1)+1;
    masPole:= DoMove(masPole, x, y);
  end;
  dgPole.Invalidate;

  //задать поле-образец:
  if Game= FullSalto then begin //- полный переворот
    //выставить на поле зелёные фишки:
    for j:= 1 to dgPole2.RowCount-1 do
      for i:= 1 to dgPole2.ColCount-1 do
        masPole2[i,j]:= WHITE
  end
  else begin //- переход к образцу
    //сделать случайные ходы:
    for i:= 1 to 7 + random(dgPole2.ColCount*dgPole2.RowCount+1-7)
do begin
```

```
      x:= Random(dgPole2.ColCount); y:= Random(dgPole2.RowCount);
      masPole2:= DoMove(masPole2, x, y);
    end;
  end;
  dgPole2.Invalidate;

  //запомнить начальную позицию:
  MemoryIn;
end; //NewPlay
```

В процедуре нажатия кнопки мыши нужно проверить, не нажата ли она на клетке с оцифровкой. В этом случае ход, разумеется, делать не следует.


```
//ПЕРЕВЕРНУТЬ ФИШКУ ИЛИ СДЕЛАТЬ ХОД
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  //нельзя делать ходы на клетки с оцифровкой:
  if ACol * ARow = 0 then exit;
  //нажата левая кнопка мыши и клавиша Shift -
  //инвертируем цвет клетки:
  if (ssLeft in shift) and (ssShift in shift ) then begin
    if masPole[ACol,ARow]= 1 then masPole[ACol,ARow]:= 0
    else masPole[ACol,ARow]:= 1;
    dgPole.Invalidate;
  end

  //нажата левая кнопка мыши без клавиши Shift -
  //делаем ход:
  else if ssLeft in shift then begin
    inc(Hod); lblHod.Caption:= 'Ход - ' + inttostr(Hod);
    masPole:=DoMove(masPole, ACol, ARow);
    dgPole.Invalidate;
    //запомнить ход:
    Moves[Hod].x:= ACol;
    Moves[Hod].y:= ARow;
    Hod_max:= Hod;
    //проверить, не решена ли задача:
    if IsReady then Ready;
  end;
end;  //dgPoleMouseDown
```

В процедуру *Prepare* добавьте только одну строку:

```
Hod_max:= 0;
```

Практически целиком придётся исправить процедуру выполнения хода, так как изменились правила игры:

```
//ВЫПОЛНИТЬ ХОД
function TForm1.DoMove(arr: TPole; x, y: integer): TPole;
var i: integer;
begin
  //инвертировать заданную клетку:
  arr[x,y]:= abs(arr[x,y]-1);
  //инвертировать ряды:
  for i:= Max(y-1,1) to Min(y+1,dgPole.RowCount-1) do //-
вертикальный
    arr[x,i]:= 1- arr[x,i];
  for i:= Max(x-1,1) to Min(x+1,dgPole.ColCount-1) do //-
горизонтальный
    arr[i,y]:= 1- arr[i,y];

  result:= arr;
end; //DoMove
```


В процедуре решения задачи мы должны учесть ряды с оцифровкой, весь остальной код изменять не нужно:

```
//НАЙТИ ПЕРЕХОД ОТ ОДНОЙ ПОЗИЦИИ К ДРУГОЙ
procedure TForm1.Solution(nStep: integer);

…

  //записать ходы в протокол:
  procedure SavePos;
  var
    n: integer;
    x, y: integer;
    s: string;
  begin
    frmProtokol.listbox1.items.add('');
    for n:= 1 to Hod do begin
      s:='';
      x:= Move[n] mod w; y:= Move[n] div w;
      frmProtokol.listbox1.items.add(inttostr(n) + '. ' + let-
ter[x]+ ' ' +inttostr(y));
    end;
    frmProtokol.listbox1.items.add('');
  end;

begin
…

//переходим к следующей клетке поля:
nextCell:
  inc(nCell[Hod]);
```

```
  while (nCell[Hod] mod w = 0) or (nCell[Hod] div w= 0) do
inc(nCell[Hod]);
…
 end;  //Solution
```

Для вывода оцифровки потребуется глобальная *константа*

```
letter= 'ABCDEFGHI';
```

Конечно, можно обойтись и без нее – в этой программе, а вот в других такой способ может и пригодиться.

Так как мы решили увеличить размеры полей до 9 x 9 клеток, то в меню нужно добавить два пункта - *mi8x8* (**Tag**=8, **ShortCut**=F8) и *mi9x9* (**Tag**=9, **ShortCut**=F9). Из-за оцифровки нам придётся подправить код в процедуре, обрабатывающей нажатие на пункты меню *mi4x4 - mi9x9*:

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЕЙ
procedure TForm1.mi4x4Click(Sender: TObject);
…
begin
…
  //размеры поля:
  n:= (Sender as TMenuItem).Tag+1;
…
end;  //mi4x4Click
```

То же самое касается и следующей процедуры:

```
//ВОССТАНОВИТЬ ЗАПОМНЕННУЮ ПОЗИЦИЮ
procedure TForm1.miMemoryOutClick(Sender: TObject);
begin
…
    case w-1 of
      4: mi4x4.Checked:= True;
      5: mi5x5.Checked:= True;
      6: mi6x6.Checked:= True;
      7: mi7x7.Checked:= True;
      8: mi8x8.Checked:= True;
      9: mi9x9.Checked:= True;
    end;
…
end;  // miMemoryOutClick
```

В процедуру *miRedoClick* нужно добавить проверку на возможность возврата хода, о чём мы говорили ранее:

```
//ОТМЕНИТЬ ПОСЛЕДНИЙ ВОЗВРАТ ХОДА
procedure TForm1.miRedoClick(Sender: TObject);
```

```
begin
…
  if Hod< Hod_max then inc(hod) else exit;
…
end;  //miRedoClick
```

И наконец, я хочу напомнить, что пункт меню *miHint* оставлен для ваших собственных изысканий, поскольку в программе отсутствует процедура вывода подсказок.

Помогаем по-новому!

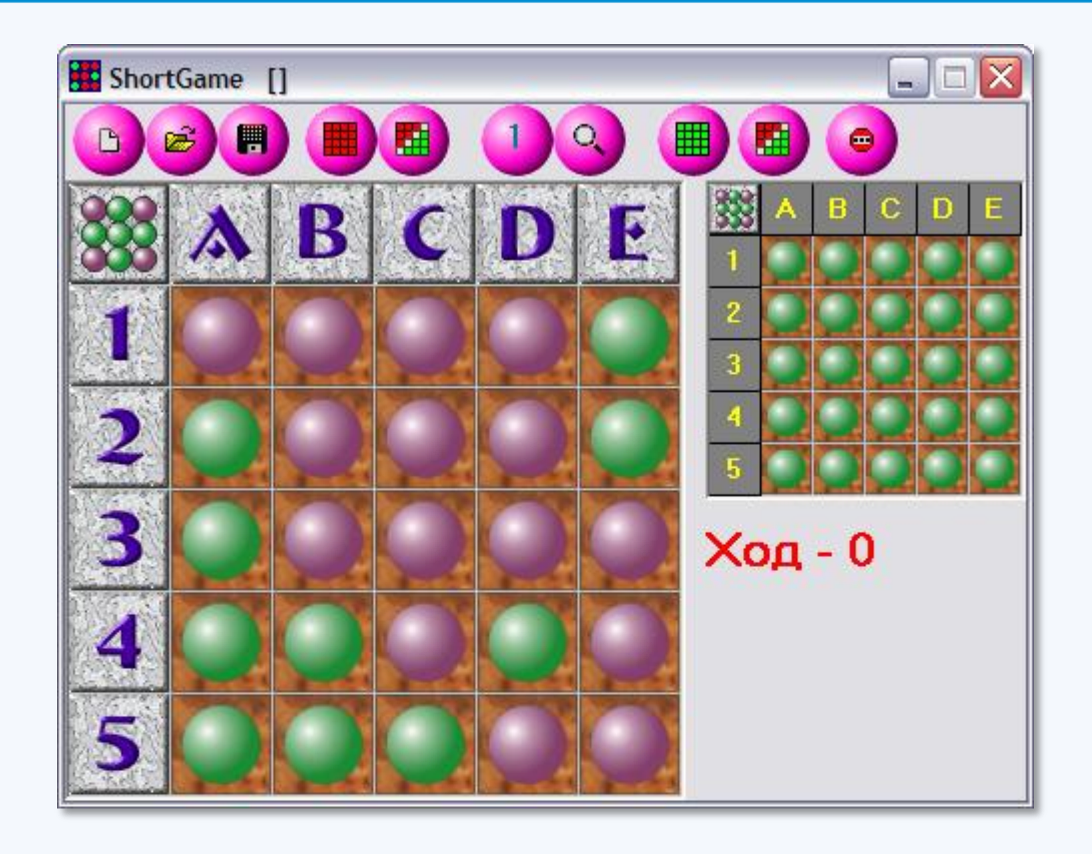
Давайте немного улучшим *помощь* для этой игры. Если файл справки достаточно длинный, то просматривать его будет не очень удобно. Поэтому лучше разбить его на отдельные разделы (темы), а на первой странице сделать небольшое *оглавление*, позволяющее быстро перейти в нужный раздел.

Прежде всего, подготовьте в текстовом редакторе файл справки (на этот раз мы воспользуемся редактором *Microsoft Word*, так как нам потребуются такие его возможности, которые могут отсутствовать в более простых программах). Примерный вид справки показан ниже:

ShortGame Help

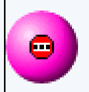
ShortGame - занимательная логическая игра-головоломка, в которой требуется все шарики на левом, игровом поле сделать зелёными (как это показано на правом, контрольном поле) за минимальное число ходов.

Ход состоит в изменении цвета любого шарика на игровом поле (зелёный шарик становится фиолетовым, фиолетовый – зелёным), но при этом изменяют цвет и все соседние по вертикали и горизонтали шарики. Чтобы выполнить ход, достаточно щёлкнуть мышкой по выбранному шарику.



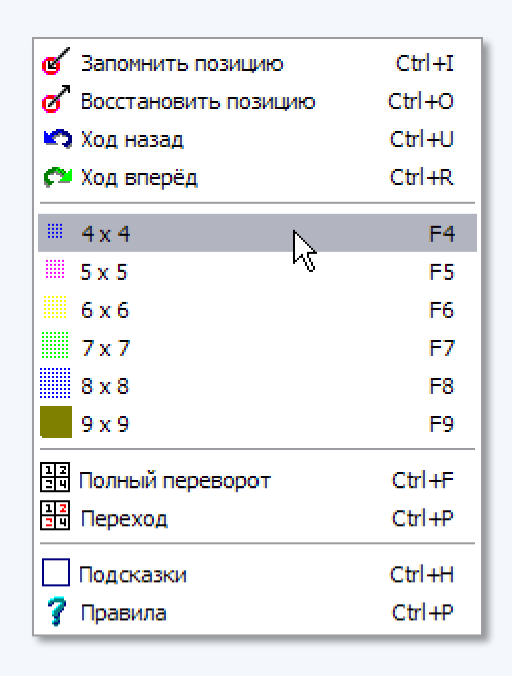
Управляющие кнопки:

	- начать новую игру, на игровое поле выводится новое задание,
	- загрузить задачу с диска,
	- сохранить позицию на диске,
	- сделать все шарики на игровом поле фиолетовыми,
	- изменить цвет всех шариков игрового поля на противоположный,
	- установить глубину поиска решения задачи,
	-найти решение задачи,
	- сделать все шарики правого поля зелёными,
	- изменить цвет всех шариков правого поля на противоположный,

 ***- остановить поиск решения задачи.***

Всплывающее меню:

Вы также можете управлять программой с помощью всплывающего меню. Чтобы вызвать его, нажмите правую кнопку мыши.



Первая группа команд поможет вам самостоятельно найти короткое решение задачи. Вы можете запомнить любую позицию на поле, а затем восстановить её, а также вернуться назад на любое число ходов (или вернуть уже сделанные ходы при откате).
Вторая группа команд устанавливает размеры полей. Разумеется, чем больше размер поля, тем труднее будет решить задачу.
У игры имеется две разновидности – полный переворот, когда все шарики на игровом поле нужно сделать зелёными, и переход, когда следует «перекрасить» шарики на игровом поле так, как на правом поле-образце. В данной версии программы реализован только полный переворот.
Если включен режим подсказки, то программа будет отмечать те клетки игрового поля, на которые нужно ходить. В данной версии программы этот режим отсутствует.

Не забудьте выделить *заголовки* с помощью стилей, чтобы они составили оглавление справки! Также между всеми разделами необходимо вставить символ конца страницы (меню *Главная|Вставка|Разрыв страницы*).

Закончив работу с текстом справки, сохраните файл с расширением *RTF*.

Теперь вы можете откомпилировать его в какой-нибудь программе, предназначенной для создания справочных файлов. Например, в

программе *Help & Manual* вы сможете получить справку в различных форматах – *HLP, CHM, EXE, HTML, PDF*.

Пусть это будет *исполняемый* файл *ShortGame_Help.exe* в папке *Help*. Тогда процедура вызова справки будет такой:

```
//ПОКАЗАТЬ ФАЙЛ СПРАВКИ
procedure TForm1.miHelpClick(Sender: TObject);
var
  SEInfo: TShellExecuteInfo;
  ExitCode: DWORD;
  ExecuteFile: string;
Begin
  hide;
  ExecuteFile := 'Help/ShortGame_Help';
  FillChar(SEInfo, SizeOf(SEInfo), 0);
  SEInfo.cbSize := SizeOf(TShellExecuteInfo);
  with SEInfo do begin
    fMask := SEE_MASK_NOCLOSEPROCESS;
    Wnd := Application.Handle;
    lpFile := PChar(ExecuteFile);
    nShow := SW_SHOWNORMAL;
  end;
  if ShellExecuteEx(@SEInfo) then begin
    repeat
      Application.ProcessMessages;
      GetExitCodeProcess(SEInfo.hProcess, ExitCode);
    until (ExitCode <> STILL_ACTIVE) or Application.Terminated;
  end
  else ShowMessage('Error!');
  show;
end;  //miHelpClick
```


При выборе пункта меню *Правила* окно приложения исчезнет с экрана и появится текст справки. В левой части окна находится список разделов, соответствующий заголовкам в тексте справки. Щёлкнув на одной из строк, вы попадёте в нужное место справки (Рис. 5.38).

После того как вы закроете окно справки, на экране вновь появится окно приложения.

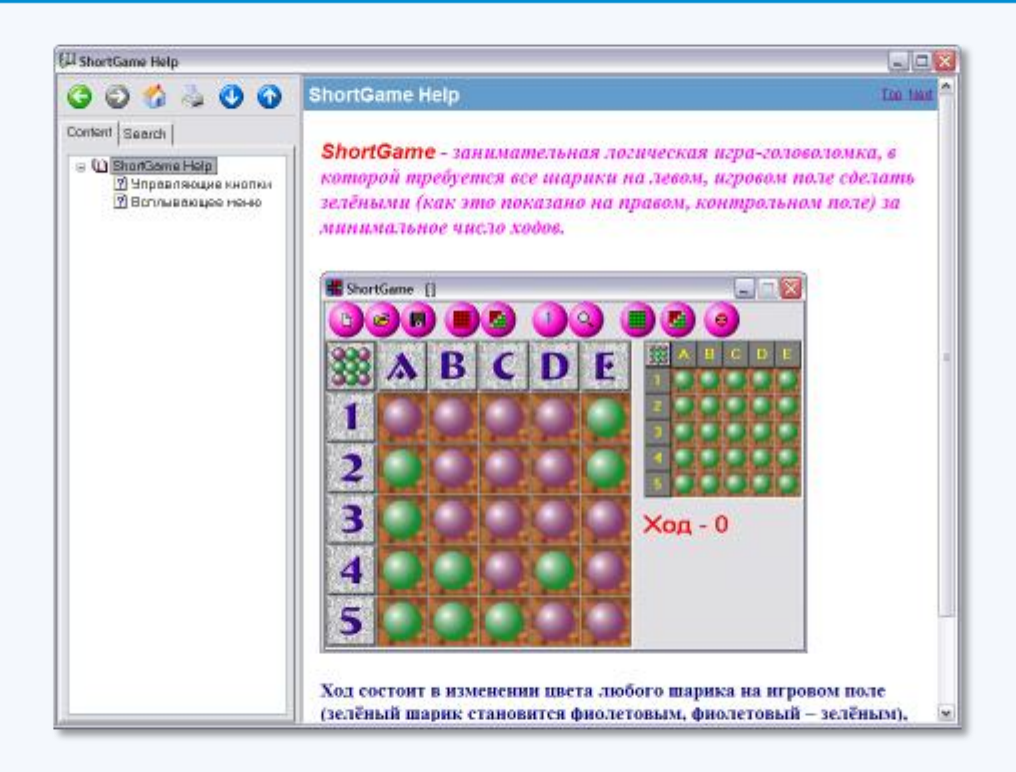


Рис. 5.38. Справка с оглавлением!



В операционной системе *Windows 7* справка работает неверно. Имейте это в виду!

Искусство побеждать

Казалось бы, правила игр *XorGame* и *ShortGame* настолько похожи, что и задачи должны решаться почти одинаково. Да не тут-то было! Ни одно из наших правил для игры *XorGame* здесь не действует, всё нужно начинать сначала!

Впрочем, стратегию игры найти нетрудно. Достаточно переворачивать *фиолетовые* фишки, начиная с верхнего ряда. Сделать это можно разными способами, и все они ведут к освобождению верхних рядов. Например, вы можете ходить последовательно в соседние справа или снизу клетки, но лучше подыскать более короткие комбинации уничтожения фиолетовых фишек.

Как бы то ни было, в итоге фишки останутся только в двух нижних рядах (легко избавиться и от предпоследнего ряда, но, если вы посмотрите на шаблон в начале раздела, то поймёте, что он занимает именно два ряда у нижней границы). Конечно, с равным успехом можно сгонять фишки к

верхней или боковой границе поля, но это ничего не изменит в решении задачи.

Оставшиеся в двух последних рядах фишки должны либо сразу составить нужную фигуру (*уголок* или букву *Т* вверх тормашками), либо легко уничтожаться, не «залезая» в верхние ряды. Для поля 4 х 4 клетки так всегда и получается, поэтому здесь вы не встретите большого сопротивления со стороны фиолетовых фишек, хотя, вероятно, вы придете к победе не самым коротким путём. А вот с другими полями всё не так просто! Произвольное уничтожение фиолетовых фишек обычно приводит к нерешаемым в пределах последних двух рядов расположениям фишек. Следовательно, переворачивать фишки в *предыдущих* рядах нужно так, чтобы получить *выигрышную* заключительную позицию. Если вам удастся найти правила выбора промежуточных ходов, то искусство побеждать станет наукой, а ваше имя навсегда останется в памяти благодарных игроков. Дерзайте!



Исходный код программы находится в папке **Short**.

Другие игры-оборотни

В игре *Flat Game* (версия 1.0, 2001 г.) (Рис. 5.39) Дмитрия Шумского вместе с «эпицентрической» клеткой изменяют цвет и все остальные клетки ряда – только вертикального или только горизонтального (а не обоих сразу, как в игре *Флип-Флоп*).

Поскольку щелчок на клетке поля не может однозначно идентифицировать ваши намерения, то щёлкать следует на *оцифровке* вокруг поля (так что она может пригодиться не только при записи ходов!). Других сложностей ни в реализации правил, ни в решении задач, как будто, нет.



Рис. 5.39. *Flat Game*

Гораздо более изощрённое правило перекрашивания клеток вы найдёте в игре *Камешки* (версия 1.7, 1999 г.) Здесь предлагается изменять цвет клеток по периметру квадрата 3 х 3 клетки (исключая центральную).

Задания выводятся последовательно из встроенной библиотеки, выбирать самостоятельно вы их не сможете. Другое неудобство игры – поле фиксированных размеров 7 х 5 клеток (Рис. 5.40).

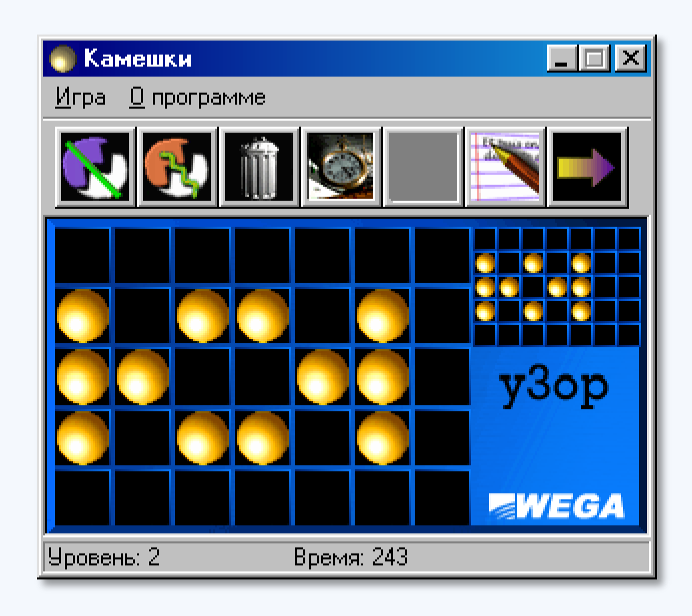


Рис. 5.40. *Камешки*

Взяв поле б***о***льших размеров и набор разных шаблонов переворачивания фишек, вы сможете получить гораздо более интересные игры.

Вот, пожалуй, и все сколько-нибудь заметные игры-оборотни (возможно, вам известны и другие). Как видите, сделано не много, так что программисты головоломок в большом долгу перед человечеством. Зато каждый из вас может отличиться на поприще создания новых игр!

Чёт и нечет

Числа правят миром.

Пифагорейское

Проверка на чётность – достаточно простой приём, который позволяет, однако, решать сложные задачи. Например, Евклид (или даже его предшественники из пифагорейской школы) с его помощью доказал иррациональность квадратного корня из двойки. Нам тоже довелось воспользоваться проверкой на чётность при программировании игры *XorGame*. Кроме того, она бывает полезна при нахождении решений перестановочных и топологических задач, а также многих головоломок.



«Проверкой на чётность» занимаются и девушки, гадающие на ромашке - «любит-не любит», и сатирик-юморист Михаил Задорнов, убеждённый, что женщинам следует дарить исключительно *нечётное* число цветов, и «менеджеры по продажам», уверенные, что покупатель с б***о***льшим удовольствием выложит 35 рублей, нежели 32.

Вообще, люди с удивительным упорством предпочитают одни числа другим, например *тройка* и *семёрка* явно предпочтительнее двойки или восьмёрки. Очень любимы числа, оканчивающие на 0 и 5 (вспомните юбилейные даты). Далее следуют числа, оканчивающиеся на 2, 3, 7, 6, 4, 9, 1. Причина тёплого отношения к «круглым» числам, конечно, заключается в десятичной системе счисления, а вот с остальными всё более загадочно. Впрочем, и природа в целом не одинаково относится к чётным и нечётным числам. Так, большинство животных (исключая иглокожих и головоногих) имеют *чётное* количество конечностей (следствие симметричности их тела), химические элементы с нечётными порядковыми номерами обычно более редки, чем соседние с ними чётные, а потому и более дороги. Теплота плавления углеводородов, начиная с гептана, имеющих чётное число атомов углерода в цепочке, заметно выше, чем у соседних с ними нечётных углеводородов. И эти примеры можно продолжать и продолжать…

Но вернёмся к нашим заботам. Для плавного перехода к теме рассмотрим шуточный *фокус* с тремя пустыми бокалами (годятся и стаканы, и другая посуда). Поставьте их на стол так, чтобы два бокала стояли ножками

(донышками) *вверх*, а между ними находился один бокал ножкой *вниз* (Рис. 5.41).



Рис. 5.41. Ножки - на ширине плеч



Необходимо за три «хода», переворачивая каждый раз *одновременно* два бокала, поставить все бокалы ножками вниз (Рис. 5.42).



Рис. 5.42. Правильное положение

Сделать это нетрудно.

1. Переворачиваем *первый* и *второй* бокалы (Рис. 5.43).



Рис. 5.43. Первый ход

2. Переворачиваем *первый* и *третий* бокалы (Рис. 5.44).



Рис. 5.44. Второй ход

3. Переворачиваем *первые два* бокала – задача решена (Рис. 5.45).



Рис. 5.45. Третий ход

Теперь незаметно переверните средний бокал ножкой вверх (Рис. 5.46)



Рис. 5.46. Хитрый трюк

и предложите кому-нибудь повторить фокус.

Если вы сравните последнюю картинку с первой, то легко заметите, что положение бокалов изменилось, и теперь головоломку невозможно решить за любое число ходов. И вот почему.

В первом случае в правильном положении (ножкой вниз) находится 1 бокал, а в конечном положении – 3. Оба эти числа *нечётные*. Во втором случае в правильном положении находятся 2 бокала – *чётное* число, а в конечном положении те же 3 бокала - *нечётное* число. Но при одновременном переворачивании двух бокалов чётность «системы» не изменяется.

Если мы перевернём из положения (Рис. 5.47)



Рис. 5.47. Новое исходное положение

два бокала ножками вверх (первый и третий), то в правильном положении будет находиться 0 бокалов – *чётное* число (будем считать, что это так). Если перевернуть другую пару бокалов, то ситуация вообще не изменится. Легко проверить, что и следующие ходы не нарушат чётности позиции, а это значит, что она никогда не станет нечётной и фокус повторить не удастся.

Если вы замените бокалы нашими фишками с двумя сторонами, то сразу поймёте, что этот фокус - всего лишь разновидность головоломок-оборотней, в которой все фишки вытянуты в одну линию.

Пусть в начальной позиции все бокалы стоят правильно – ножками вниз (Рис. 5.48)

,

Рис. 5.48. Правильное положение

а в конечной – все ножками вверх (Рис. 5.49).



Рис. 5.49. Конечное положение

Первая позиция *нечётная*, вторая – *чётная*, так что «фокус не удастся». Но попробуем решить *общую* задачу (см. журнал *Наука и жизнь*, №10 за 1982 год). Пусть имеется *m* бокалов ножками вниз, из которых за один раз можно перевернуть *n* любых бокалов. Требуется за минимальное число ходов получить позицию, в которой все бокалы находятся ножками вверх. Например, решим такую задачу: *m = 7, n = 5* (Рис. 5.50).



Рис. 5.50. Задачка!

Для удобства мы напишем небольшую программу для решения таких головоломок, ведь они могут быть интересны и вашим знакомым.

«Ножки вверх!», или Головоломка с бокалами

Начните новый проект. Подготовьте картинки с бокалами в двух положениях и двух цветов. Как обычно, начните с подготовки *формы* (Рис. 5.51).

```
  Left = 14
  Top = 119 (расположите форму на экране так, чтобы были видны все
бокалы)
  Width = 770
  Height = 238 (размеры формы будут подгоняться под компоненты при
```

```
запуске программы, поэтому сильно не старайтесь)
  BorderIcons = [biSystemMenu]
  Caption = ' Бокалы'
  Color = clWhite
  Icon.Data – нарисуйте значок программы
  OnCreate = FormCreate
```

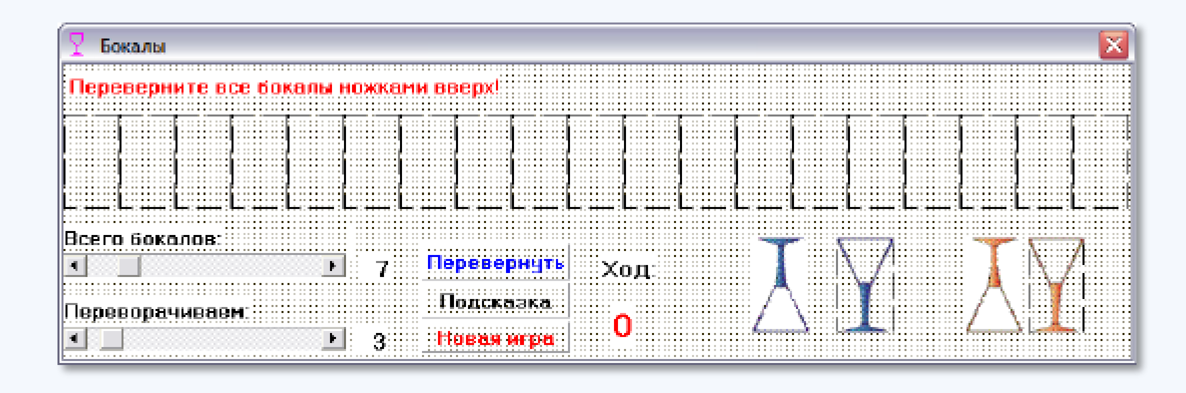
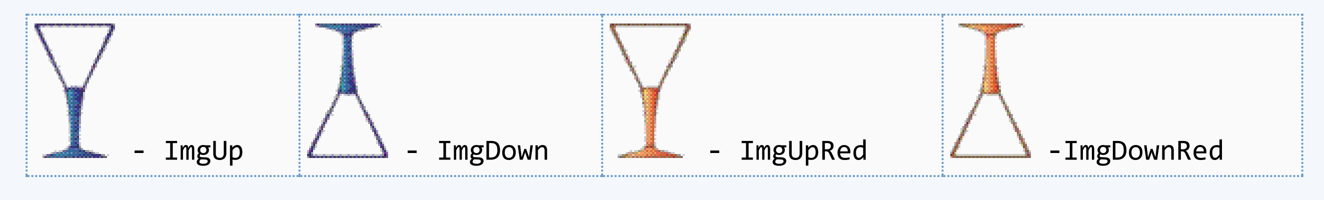



Рис. 5.51. Интерфейс приложения

Поместите картинки на хранение в компоненты *TImage*:




```
    Width = 40
    Height = 67 (размеры картинок)
    AutoSize = True
    Transparent = True
    Visible = False
```

«Игровые» бокалы загрузите в компоненты *Image1.. Image19*:

```
    Tag = 1, 2, 3, …
    Left = 0, 40, 80, …
    Top = 36
    Width = 40
    Height = 67
    AutoSize = True
    Transparent = True
    OnClick = Image1Click (для всех компонентов).
```

Над бокалами установите метку *Label2* с напоминанием:

```
    Left = 4
    Top = 8
    Width = 308
    Height = 16
```

```
Caption = 'Переверните все бокалы ножками вверх!'
Font.Color = clRed    Font.Style = [fsBold]
```

Изменять количество бокалов на «столе» мы будем с помощью *скроллера sbBok* типа *TScrollBar* (вы можете найти и более изящное решение, если сочтёте нужным):

```
Left = 0
Top = 136
Width = 201
Height = 16
Max = 19 (максимальное число бокалов)
Min = 5   (минимальное число бокалов)
Position = 7 (значение по умолчанию)
OnChange = sbBokChange
```

Текущее значение числа бокалов можно увидеть в *метке lblBok*:

```
Left = 212
Top = 136
Width = 21
Height = 20
Alignment = taRightJustify
AutoSize = False
Caption = '7'
Font.Height = -16
Font.Style = [fsBold]
```

Аналогично мы будем задавать и число одновременно переворачиваемых бокалов *скроллером sbPerev*:

```
Left = 0
Top = 188
Width = 201
Height = 16
Max = 18
Min = 2
Position = 3
OnChange = sbPerevChange
```

и показывать в *метке lblPerev*:

```
Left = 208
Top = 188
Width = 23
Height = 20
Alignment = taRightJustify
AutoSize = False
Caption = '3'
Font.Height = -16
Font.Style = [fsBold]
```

Общее число бокалов мы будем хранить в глобальной *переменной*

```
  //всего бокалов:
  nBok: integer;
```

А число одновременно переворачиваемых бокалов – *в переменной*

```
  //переворачиваем за один раз:
  nPerev: integer;
```


При перемещении ползунка скроллера *sbPerev* в метке *lblPerev* выводится новое значение числа переворачиваемых бокалов, а также оно запоминается в переменной *nPerev*. Так как число переворачиваемых бокалов должно быть *меньше* общего числа бокалов, то значение *nPerev* устанавливается в допустимых пределах:

```
//ИЗМЕНИТЬ ЧИСЛО ПЕРЕВОРАЧИВАЕМЫХ БОКАЛОВ
procedure TForm1.sbPerevChange(Sender: TObject);
begin
  lblPerev.Caption:= inttostr(sbPerev.Position);
  nPerev:= sbPerev.Position;
  while nPerev >= nBok do sbPerev.Position:= sbPerev.Position-1;
  NewPlay;
end;  //sbPerevChange
```

Процедура обработки изменения позиции ползунка второго скроллера ещё проще:

```
//ИЗМЕНИТЬ ЧИСЛО БОКАЛОВ
procedure TForm1.sbBokChange(Sender: TObject);
begin
  lblBok.Caption:= inttostr(sbBok.Position);
  nBok:= sbBok.Position;
  //скорректировать число переворачиваемых бокалов:
  sbPerevChange(Self);
end; //sbBokChange
```

После любого изменения условия задачи следует начинать *новую* игру, так как все предыдущие ходы в этом случае теряют всякий смысл. В процедуре подготовки к новой игре снимаются значения со скроллеров и обнуляется число ходов:

```
//НОВАЯ ИГРА
procedure TForm1.NewPlay;
begin
    //число бокалов:
  lblBok.Caption:= inttostr(sbBok.Position);
  nBok:= sbBok.Position;
  ChangeBok;
```

```
  //переворачиваем за 1 раз:
  lblPerev.Caption:= inttostr(sbPerev.Position);
  nPerev:= sbPerev.Position;
  //обнулить ходы:
  Hod:= 0;
  lblHod.Caption:= '0';
end; //NewPlay
```

Бокалы могут находиться в одном из следующих *положений*:

Up – «горлышком» вверх,
Down – «горлышком» вниз,
UpRed – «горлышком» вверх, выделенное состояние,
DownRed – «горлышком» вниз, выделенное состояние,
None – бокал с заданным номером отсутствует.

Для описания положения бокалов введём новый *тип*

```
type stBok= (Up, Down, UpRed, DownRed, None);
```

и объявим глобальный *массив*

```
  //массив бокалов:
  masBok: array[1..19] of stBok;
```

В начале игры первые *nBok* бокалов находятся в правильном положении *Up*, а все остальные бокалы от *nBok+1* до *19* вообще в игре участия не принимают (*None*):

```
//ПЕРЕВЕСТИ БОКАЛЫ В ИСХОДНОЕ ПОЛОЖЕНИЕ
procedure TForm1.ChangeBok;
var
  i: integer;
begin
  for i:= 1 to 19 do begin
    if i <= nBok then
      masBok[i]:= Up
    else
      masBok[i]:= None;
  end;
  DrawBok;
end; //ChangeBok
```

Сначала все изменения на «столе» заносятся в массив *masBok*, а затем отображаются на экране в процедуре *DrawBok*. В ней последовательно перебираются все (в том числе и невидимые) компоненты *TImage* с бокалами и, соответственно значению в массиве *masBok*, в каждый компонент выводится нужная картинка:

```
//ПОКАЗАТЬ БОКАЛЫ
procedure TForm1.DrawBok;
var
  i: integer;
  comp: TComponent;
begin
  for i:= 1 to 19 do begin
    comp:= FindComponent('Image'+inttostr(i));
    (comp as Timage).top:= 36;
    case masBok[i] of
      None: (comp as Timage).visible:= False;
      Up: begin
            (comp as Timage).visible:= True;
            (comp as Timage).Picture.Assign(ImgUp.Picture);
          end;
      Down: begin
            (comp as Timage).visible:= True;
            (comp as Timage).Picture.Assign(ImgDown.Picture);
          end;
      UpRed: begin
            (comp as Timage).visible:= True;
            (comp as Timage).Picture.Assign(ImgUpRed.Picture);
          end;
      DownRed: begin
            (comp as Timage).visible:= True;
            (comp as Timage).Picture.Assign(ImgDownRed.Picture);
          end;
    end;
  end;
end; //DrawBok
```


Число выполненных ходов хранится в глобальной *переменной*

```
  //ходы:
  Hod: integer=0;
```

А для вывода числа ходов нам потребуются *метки Label1*:

```
    Left = 384
    Top = 136
    Width = 40
    Height = 20
    Caption = 'Ход:'
    Font.Height = -16
    Font.Style = [fsBold]
```

и *lblHod:*

```
    Left = 392
    Top = 172
    Width = 15
```

```
Height = 29
Alignment = taCenter
Caption = '0'
Font.Color = clRed
Font.Height = -24
Font.Style = [fsBold]
```

При запуске программы устанавливается в *True* свойство формы *AutoSize*, чтобы её размеры при любом числе бокалов были оптимальными, а затем выводится *начальная* позиция, определяемая положением движков скроллеров:


```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  form1.AutoSize:= True;
  NewPlay;
end; //FormCreate
```

Поскольку вы можете переворачивать одновременно *любые* бокалы (но не любое их количество!), то их нужно уметь *выделять*. Для этого удобно просто щёлкать на нужных бокалах - их цвет будет меняться с *синего* (положение *Up* и *Down*) на *красный* (*UpRed* и *DownRed*), и наоборот:

```
//ВЫДЕЛИТЬ БОКАЛ
procedure TForm1.Image1Click(Sender: TObject);
var i: integer;
begin
  //номер бокала:
  i:= (Sender as TImage).tag;
  case masBok[i] of
    Up: masBok[i] := UpRed;
    Down: masBok[i] := DownRed;
    UpRed: masBok[i] := Up;
    DownRed: masBok[i] := Down
  end;
  DrawBok;
end; //Image1Click
```

Не забудьте обработчики события *OnClick* всех компонентов *TImage* связать с этой процедурой!

Сам ход выполняется при нажатии на *кнопку* *sbtPerev* типа *TSpeedButton*:

```
Left = 256
Top = 128
Width = 105
Height = 26
Cursor = crHandPoint
Caption = 'Перевернуть'
```

```
    Flat = True
    Font.Color = clBlue
    Font.Style = [fsBold]
    OnClick = sbtPerevClick
```



```
//ПЕРЕВЕРНУТЬ БОКАЛЫ
procedure TForm1.sbtPerevClick(Sender: TObject);
var i, n: integer;
begin
  //проверить ход:
  n:= 0;
  for i:= 1 to 19 do
    if (masBok[i]= UpRed) or (masBok[i]= DownRed)
      then inc(n);
  if n <> nPerev then begin
    application.MessageBox('Столько бокалов вам не перевернуть!',
NAME_PROG,IDOK);
    exit
  end;
  //выполнить ход:
  for i:= 1 to 19 do
  case masBok[i] of
    UpRed: masBok[i] := Down;
    DownRed: masBok[i] := Up
  end;
  DrawBok;
  inc(Hod);
  lblHod.Caption:= inttostr(Hod);

  //все бокалы перевёрнуты?
  n:= 0;
  for i:= 1 to 19 do
    if masBok[i]= Down
      then inc(n);
  if n = nBok then begin
    application.MessageBox('Отличная работа!', NAME_PROG,IDOK);
    NewPlay;
  end;
end; //sbtPerevClick
```

Сначала нужно проконтролировать правильность выполнения хода – число выделенных бокалов должно в точности равняться *nPerev*. Если всё верно, выделенные бокалы переворачиваются, и метод проверяет, не решена ли задача.

В любой момент вы можете начать *новую* игру, нажав *кнопку sbtNewPlay*:

```
    Left = 256
    Top = 184
    Width = 105
    Height = 22
```

```
Cursor = crHandPoint
Caption = 'Новая игра'
Flat = True
Font.Color = clRed
Font.Style = [fsBold]
OnClick = sbtNewPlayClick
```

```
//НАЧАТЬ НОВУЮ ИГРУ
procedure TForm1.sbtNewPlayClick(Sender: TObject);
begin
  NewPlay;
end; //sbtNewPlayClick
```


Как решить задачу?

Давайте вспомним о нашей задаче с семью бокалами, которые нужно переворачивать по три штуки за один раз.

Если вы станете бессистемно вертеть бокалы, то рано или поздно вы, конечно, эту задачу решите (не такая уж она и сложная), да что толку – со следующей вы будете мучиться ничуть не меньше. Лучше сразу всё хорошенько обдумать, а затем играть «со смыслом»!

Прежде всего, вы должны для себя отметить, что, сколько бы раз вы бокалы ни переворачивали, всё равно общее число перевёрнутых бокалов будет кратно трём (мы ведь переворачиваем по три бокала!). Но семь на три не делится, поэтому просто так их и не перевернуть. Легко за два раза перевернуть 6 бокалов, но тогда останется только один «неправильный» бокал, и вместе с ним нам придётся перевернуть и два правильных. Они станут неправильными и, возвращая их в нужное положение, мы опять прихватим один правильный бокал. В общем, играть нужно не так!

Первый ход очевиден – мы должны перевернуть ножками вверх любые три бокала (Рис. 5.52).



Рис. 5.52. Первый ход

Столь же очевиден и *последний* ход – нам придётся перевернуть ножками вверх какие-то три бокала (Рис. 5.53).

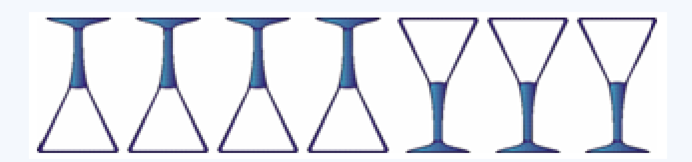


Рис. 5.53. Последний ход

Все сложности - в *промежуточных* ходах, точнее, даже в одном ходе. Попробуем его вычислить.

Ясно, что за два хода задачу не решить, так как мы сумеем перевернуть только 6 бокалов. За три хода мы перевернём 9 бокалов, а нужно 7. Но 7 – это те бокалы, которые стояли вначале ножками *вниз*, а ведь после первого хода появляются и бокалы, стоящие ножками *вверх*. Если один из них вернуть в исходное положение (ножкой вниз), то по ходу решения мы перевернём уже 8 бокалов. Но мы обязаны перевернуть его ещё один раз, иначе он так и будет стоять ножкой вниз! Таким образом, при этих операциях мы добавили к семи переворачиваемым бокалам ещё два. Итого получилось ровно 9, что и требуется!

Теперь всё предельно ясно: делая *второй* ход, мы должны возвратить в исходное положение один из тех бокалов, что был перевёрнут на первом ходу (Рис. 5.54).



Рис. 5.54. Второй ход

В расчётах мы не ошиблись – осталось ровно *три* бокала ножками *вниз*, которые и нужно перевернуть на *третьем* ходу.

Решим более сложную задачу. Пусть теперь имеется 17 бокалов, а переворачивать нужно по 5 бокалов одновременно. В лучшем случае задачу можно решить за 4 хода, перевернув 20 бокалов. То есть к 17 уже имеющимся следует добавить ещё три. А это невозможно: мы показали, что дополнительные бокалы берутся из уже перевёрнутых ножками вверх, при этом любой из них придётся переворачивать два раза. Но три на два не делится, поэтому никакие ухищрения не помогут. Придётся накинуть ещё один ход, тогда мы перевернём 25 бокалов, из которых 25 – 17 = 8

«лишних». Их могут дать 8 : 2 = 4 возвращаемых в исходное положение бокалов.

Итак, для решение задачи потребуется 5 ходов. На первом и последнем мы будем переворачивать по 5 бокалов, стоящих ножками вниз. А вот во время оставшихся трёх ходов нам и следует возвращать в исходное положение 4 бокала. Причём совершенно безразлично, как вы разделите эти бокалы между ходами – 4+0+0, 3+1+0, 2+2+0, … Стало быть, задача имеет *несколько* вариантов решения длиной в 5 ходов.

Нам осталось изложить открытый нами *алгоритм* на «дельфийском» языке и препоручить *кнопке sbtHint*:

```
Left = 256
Top = 156
Width = 105
Height = 25
Caption = 'Подсказка'
Flat = True
Font.Style = [fsBold]
OnClick = sbtHintClick
```

всю заботу о выводе подсказки:

```
//ПОДСКАЗКА
procedure TForm1.sbtHintClick(Sender: TObject);
begin
  Hint;
end;

//ВЫВЕСТИ ПОДСКАЗКУ
procedure TForm1.Hint;
var
  i, n, m: integer;
begin
  //если число бокалов нечётное, а переворачиваем чётное
  //число бокалов, то задача неразрешима:
  if odd(nBok) and not odd(nPerev) then
  begin
    showmessage('Задача решений не имеет!');
    exit
  end;

  m:= nBok;
  n:= nPerev;
  i:= 0;
  while (n*i< m) or ((n*i - m) mod 2 <> 0) do inc (i);

  showmessage('На решение задачи потребуется ходов - '+ inttostr
 (i)+ #10#13+                'Число возвращаемых бокалов = '+
```

```
inttostr ((n * i - m) div 2));
end;  //Hint
```

Вычисления проводятся в цикле

```
while (n*i< m) or ((n*i - m) mod 2 <> 0) do inc (i);
```

Он совершается до тех пор, пока число, кратное числу переворачиваемых бокалов (*n*i*), меньше общего числа бокалов или разность (*n*i - m*) не делится на два. О необходимости (и достаточности) выполнения этих условий мы только что говорили.

Теперь вы сможете не только успешно переворачивать бокалы, но и делать это непринуждённо!

Конечно, вам уже кажется, что нет таких задач, с которыми вы бы не справились. Тогда попробуйте такую.

В ряд расставлены 6 бокалов, причём первые три с вином, последние три пустые (Рис. 5.55). Вы можете взять в руку только *один* бокал. Что нужно сделать, чтобы бокалы с вином и пустые чередовались?

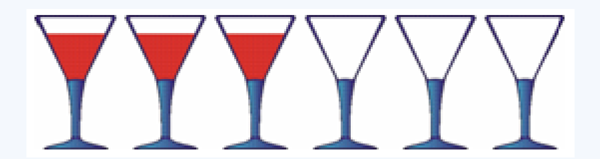


Рис. 5.55. Невинная винная задача

Ответ вы найдете на следующей странице.



1. Можно ли решить «бокальную» задачу при условии, что допускается одновременно переворачивать только идущие **подряд** бокалы? Тривиальные сочетания **m** и **n** типа 6 и 3 не учитывайте.

2. Условие то же, что и в предыдущем пункте, но бокалы располагаются кольцом, то есть первый и последний бокалы стоят *рядом*.

3. *Вероятно,* задача неразрешима при условиях 1 и 2 (попробуйте доказать или опровергнуть это предположение!), поэтому поищите задачи, в которых начальное положение бокалов отличается от стандартного (все ножками вниз). Очевидно, что существует множество таких задач (их легко получить из конечной позиции, делая ходы тем или иным способом). Вопрос в том, могут ли при этом

возникнуть *интересные* расположения бокалов или *необычные* последовательности ходов.

Ответ неожиданно прост (быстро ли вы до него додумались?) – нужно перелить вино из *второго* бокала в *пятый* и вернуть его на место. Оп-ля!



Исходный код программы находится в папке **Bokaly**.

Факультатив 3. Японский кроссворд, или Восхождение на Фудзияму мысли

Банзай!

Клич японских фанатов оригами

Японцы, конечно, молодцы – придумали такое количество отличных головоломок, что о них можно написать целое собрание сочинений! Впрочем, наши помыслы не устремляются так далеко, поэтому мы ограничимся только одной из них, но весьма интересной и полезной для ума. Самое удивительное в ней то, что эта забава, которую мы называем *Японским кроссвордом* (а то и сканвордом!) или *Японской головоломкой* не имеет никакого отношения к словесным задачам – в ней вообще нет никаких слов.



Сами японцы называют её *Edel*. Поскольку головоломка очень известна во все мире, то у нее, как и у закоренелого рецидивиста, имеются «клички» - *Нонограмма* и *Hanjie*.

Это и понятно: японцы пишут иероглифами, то есть целыми словами, из которых кроссворд составить довольно затруднительно, что, собственно, и послужило толчком для создания этой головоломки. Сначала в заданиях требовалось найти загаданный иероглиф, но потом стали использовать и обычные чёрно-белые картинки. В России, конечно, распространён именно последний вариант головоломки…

Просветление, или Что придумали японцы

-Вы неправильно коня поставили,
товарищ гроссмейстер,- залебезил одноглазый.
– Конь так не ходит.

Ильф и Петров,

Двенадцать стульев

Прежде чем перейти к самому интересному (а этим для нас, безусловно, является решение головоломок с помощью компьютера), нужно разобраться в сущности японского кроссворда.

Все подобные задачи имеют вот такой вид (Рис. 3.1)

				A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
				3	2	1	10	11	3	3	3	8	8	3	3	3	11	1	4	
					2				3	3	3	2	2	3	3	3				
									3	2	2			2	2	2				
0																				1
1	1	1																		2
1	3	2																		3
1	11																			4
2	11	1																		5
5	6	3																		6
2	2	1	1																	7
3	2	2	1																	8
11																				9
11																				10
2	6	1																		11
2	2																			12
3	2																			13
7																				14
4																				15

Рис. 3.1. Японский кроссворд

Большой белый прямоугольник, разделённый на квадратные клетки, - это *игровое поле*. Сначала оно пустое, то есть все клетки *белые*. Рисунка на нём, как на ещё не проявленной фотографии, не видно. А чтобы он «проявился», часть клеточек нужно *закрасить*. Любой непрерывный горизонтальный или вертикальный ряд чёрных клеток называется *группой*. Друг от друга группы отделяются по крайней мере одной *белой* клеткой (белые клетки могут находиться и по краям строки или столбца на игровом поле, то есть до первой и за последней группой).

Сверху и слева от игрового поля размещаются поля с числами (на нашем рисунке они выделены *серым* цветом). Мы будем называть их *числовыми полями*. Числа показывают длину каждой группы соответствующего ряда, поэтому в каждом ряду на игровом поле столько групп, сколько чисел в числовом поле. Например, на Рис. 3.2 ряд содержит 3 группы чёрных клеток – в левом числовом поле находится 3 числа. В первой группе 2 клетки, во второй – 6, в третьей – 1, то есть первое число слева (или сверху – для верхнего числового поля) задаёт длину первой группы, второе – второй и так далее. Положение групп в ряду может и отличаться от того, что на рисунке, - всё зависит от условия задачи.



Рис. 3.2. Ряд чисел и клеток



Надо заметить, что, не нарушая последовательности чисел, их можно записывать либо от левого (верхнего), либо от правого (нижнего) края числового поля. Чаще встречается второй вариант, что нередко приводит к ошибкам при решении задачи, так как начальные группы рядов располагаются на разных уровнях. Именно поэтому мы всегда будем придерживаться второго варианта.

Для наглядности дальнейших рассуждений все ряды игрового поля обозначены числами и буквами, как в шахматах (правда, у шахматистов принято нумеровать горизонтали снизу вверх, а нам удобнее делать это сверху вниз, что более естественно). В реальных задачах этот «элемент оформления» отсутствует.

Таким образом, для каждого ряда игрового поля нам известно: число групп в нём и длина каждой из групп. Дело за малым (этот малый, конечно, вы) – правильно «расставить» белые клетки. Не надейтесь, что вам удастся сделать это играючи, а лучше внимательно прочитайте правила решения японских кроссвордов и разберитесь в примере решения одного из них, не самого сложного, но и не самого простого. Даже если ваша рука и не тянется к перу, а перо – к бумаге, всё равно наши экзерсисы пригодятся вам при работе над грядущей программой.

Правила хорошего моветона

Лошадью ходи!

Один из дурацких Фединых советов

Начинать решение следует с тех рядов, которые сами в руки просятся. Думать придётся много, но это впереди.

Золотые правила решения японских кроссвордов

Правило 1: *Пустой ряд.* Перво-наперво поищите ряды (горизонтальные и вертикальные), в которых стоит только число *0*. Такая ситуация встречается нечасто, но нужно быть готовым ко всему. Так вот, это число неопровержимо свидетельствует о том, что *весь ряд состоит только из белых клеток* (в печатных изданиях все клетки изначально белые, поэтому «обелить» их не удастся, придётся прибегать ко всяческим уловкам, чтобы их отметить, - например, точками или крестиками). Найдя *0*, вы смело можете поставить на этом ряде крест (Рис. 3.3).

Рис. 3.3. Пустой ряд из белых клеток

Так как весь ряд разгадан, то слева от него мы ставим *галочку*, дабы он более не обременял нас размышлениями.

Правило 2: *Полный ряд*. Значительно чаще в задачах попадаются ряды, которые также описываются единственным числом, в точности совпадающим с длиной ряда на игровом поле. Тут, как говорится, без вариантов – весь ряд нужно «очернить». Действуем аналогично первому случаю, но в негативном исполнении (Рис. 3.4).



Рис. 3.4. Полный ряд черных клеток

Первые два правила просты и очевидны, как очевидно и то, что с их помощью можно решить только *Чёрный квадрат* Малевича или его бледного родственника. В японских кроссвордах такие шедевры не встречаются, поэтому перейдём к более сложным комбинациям клеток.

Правило 3: *Полная сумма*. Это третья – и последняя – возможность разом покончить с целым рядом. Так как между группами чёрных клеток должна находиться хотя бы одна белая, то все они займут как минимум столько клеток, сколько в сумме составляют все группы, плюс по одной клетке между ними (то есть число групп в ряду за вычетом единицы). Если мы найдём ряд с такой суммой чисел, то его участь предрешена – он будет беспощадно раскрашен (Рис. 3.5).



Рис. 3.5. Полная сумма

При этом нам потребуется уже известная сноровка, но это приятные хлопоты. Как и раньше, отмечаем ряд галочкой. Третьим правилом придётся пользоваться довольно часто, но и его маловато будет для решения даже самой простой задачи.



Ясно, что *Правило 2* является частным случаем более общего *Правила 3*. При ручном разгадывании кроссворда этот факт ничего не даёт, но при программировании он нам ещё послужит.

Правило 4: *Длинная группа*. Больше нам не удастся решить целиком весь ряд, придётся брать частями (подобный вид дележа чужих денег был не по вкусу Остапу Бендеру, он предпочитал всё сразу, но мы вынуждены идти на компромисс). Если в числовом поле стоит *единственная группа*, длина которой больше половины длины ряда на игровом поле, то, как бы ни располагалась эта группа, часть клеток в середине ряда всегда будет занята чёрными клетками. Эти клетки вычисляются очень просто – достаточно отсчитать от края игрового поля столько клеток, сколько должен занять этот ряд, и посмотреть, сколько клеток останется ещё не закрытыми с противоположного края. Так как разгаданные чёрные клетки должны располагаться симметрично относительно границ ряда, то столько же неразгаданных клеток должно быть и в начале ряда. Число этих клеток можно узнать, если из длины ряда вычесть длину группы. Мы пока не знаем, какого цвета будут граничные клетки, а вот остальные обязательно будут *чёрными* (Рис. 3.6).



Рис. 3.6. Длинная группа

Чтобы лучше понять, почему так происходит, нарисуем группу в *крайних* положениях (вы можете сделать то же самое и для всех промежуточных положений группы, но это ничего не изменит) (Рис. 3.7).

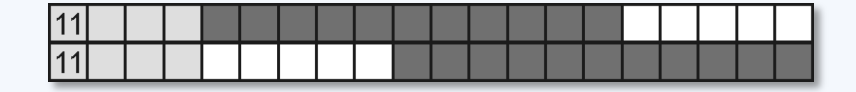


Рис. 3.7. Нахлест

Отчётливо видно, что первые и последние 5 клеток ряда (16 – 11 = 5) могут быть как чёрными, так и белыми, поэтому мы не можем пока однозначно определить их цвет. А вот 6 клеток в центре группы (11 – 5 = 6) обязательно будут *чёрными*, как ни крути!

Это правило следует хорошенько запомнить, так как пользоваться им придётся очень часто.

Его действие можно распространить и на тот случай, когда в ряду не одна, а *две* группы. Вычтите из длины ряда длину одной из групп и ещё одну клетку для «зазора» между группами. Если длина второй группы больше половины полученной разности, то часть клеток второй группы обязательно будет *чёрной*. Например, если длина ряда 9 клеток и в нём две группы из 4 и 3 клеток, то даже в крайнем правом положении вторая группа «выбьет» из ряда 4 клетки. На долю первой группы останется 5 клеток,

и мы легко можем подсчитать, что 3 клетки ряда, начиная со второй, обязаны быть *чёрными* (Рис. 3.8).

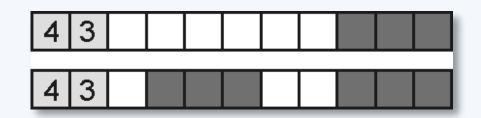


Рис. 3.8. Двойной нахлест

Кстати говоря, правила составления классических японских кроссвордов запрещают и пустые, и полные строки, и полные суммы, но у нас на это дело - по привычке - смотрят шире, чем в «узкоглазой» Японии, то есть у нас можно всё. И ещё о запретном: размеры игрового поля в классическом кроссворде должны быть кратны пяти. Но судьба этого запрета в России столь же печальна, как и предыдущих...

Железные правила решения японских кроссвордов

Первые два правила можно применить только однократно, в самом начале игры, третье и четвёртое могут пригодиться и дальше, если учитывать уже разгаданные клетки ряда. Все следующие правила нужно проверять после *каждого* изменения ситуации на игровом поле.

Правило 5: *Нахлёст*. Если в ряду осталась только *одна* неразгаданная группа, длина которой больше половины оставшихся неразгаданными клеток, то часть клеток в середине группы обязательно будет *чёрными*. Их количество и положение легко установить, пользуясь *Правилом 1* (Рис. 3.9).

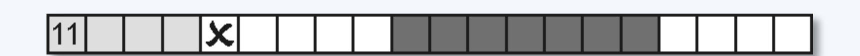


Рис. 3.9. Нахлест

Правило 6: *Габариты.* Важно отмечать не только заведомо чёрные, но и заведомо *белые* клетки. Проверяйте, какие клетки наверняка не могут быть чёрными, и, значит, они белые. Такое положение часто возникает, если в ряду уже отгаданы чёрные клетки и мы можем точно установить предельные клетки чёрной группы. На верхнем рисунке (Рис. 3.10) показана позиция на игровом поле, когда в ряду уже разгадана одна чёрная клетка. Если предположить, что она является начальной для чёрной группы, то, отсчитав влево 11 клеток, мы убедимся, что последние две клетки не могут входить в эту группу, поэтому они непременно *белые* (Рис. 3.10).

Рис. 3.10. Габариты

Правило 7: *Слитная группа*. Если в ряду осталась неразгаданной только *одна* группа и имеются *чёрные* клетки, разделённые неразгаданными, то все они принадлежат одной группе и все неразгаданные клетки между чёрными тоже *чёрные* (Рис. 3.11).

Рис. 3.11. Слитная группа

Правило 8: *Автогруппа*. Проверяйте, не образовались ли после закрашивания клеток в одних рядах полные группы в других. Если необходимо, обрамляйте их с двух сторон *белыми* клетками (Рис. 3.12).

Рис. 3.12. Автогруппа

Правило 9: *Граничные клетки*. Если на поле есть *чёрная* клетка, граничащая с *белой*, то она является либо начальной, либо конечной клеткой одной из групп ряда (Рис. 3.13).

Рис. 3.13. Граничные клетки

Правило 10: *Вилка*. Если между двумя белыми клетками (или между белой клеткой и краем поля) не может разместиться ни одна группа, то все клетки между ними *белые* (Рис. 3.14).

Рис. 3.14. Вилка

Теперь вы вполне сможете «правильно» решать задачи. Но имейте в виду, что невозможно описать все положения, могущие возникнуть на игровом поле, поэтому подходите к применению правил творчески. К примеру, многими из них можно воспользоваться и в более сложных случаях, чем это описано.

Решительный момент, или Берём быка за бока

Тореадор, смелее в бой!

Оперный призыв

Давайте решим для примера задачу, которую вы уже видели в начале главы.

1. Следуя нашим правилам, мы легко обнаруживаем **ряд 1**, в котором для нас заботливо поставлен *нолик*, и решаем его без всяких усилий (**Правило 1**).
2. К сожалению, на этом блага цивилизации заканчиваются, так как никто не позаботился воплотить **Правило 2** в жизнь. Зато – мир не без добрых людей! – **Правило 3** представлено во всей красе: сразу два ряда – **5** и **6** - подчиняются ему, так что мы с лёгким сердцем можем разом покончить с обоими.
3. В столбцах **N** и **E** притаились *длинные* ряды. Так как верхняя клетка уже известна (она *белая*), то группу из 11 клеток нужно разместить на 14 клетках. По **Правилу 5**, определяем, что 8 клеток, 5-12, *чёрные*.
4. **D** – опять *длинный* ряд. Верхняя клетка разгадана, и группа из 10 клеток должна разместиться на тех же 14 клетках. Если отсчитать 10 клеток сверху (от белой) и снизу (от края поля), то окажется, что 7 клеток, 5-11, в середине ряда *чёрные*. По **Правилу 6**, отсчитываем вниз от **D5** 10 клеток и выясняем, что последняя клетка ряда - **D15** - может быть только *белой*.

После всех манипуляций игровое поле должно выглядеть так (Рис. 3.15).

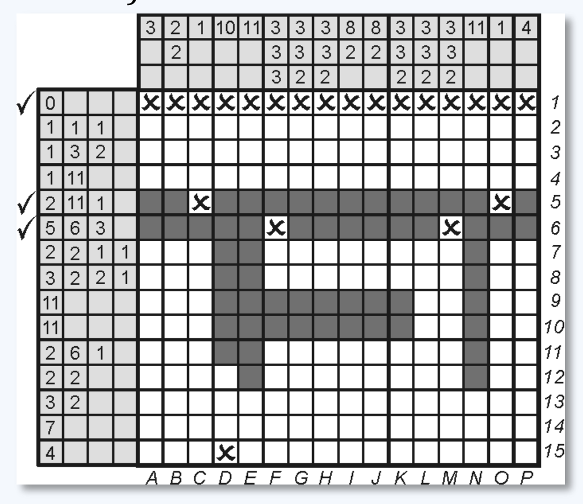


Рис. 3.15. Четыре шага позади

5. **Строки 9, 10**. Воспользуемся **Правилом 7** и закрасим клетки **L**, **M**. Так как теперь группы на поле состоят из 11 клеток, что равно заданной длине групп, то эти группы решены. Других групп в этих рядах нет, поэтому весь ряд решён. Оставшиеся клетки помечаем как *белые*.
6. **Столбцы F, M.** Клетка **5** может быть только *конечной* для первой группы из 3 клеток, поэтому можно закрасить клетки **3** и **4**. Группа разгадана, отмечаем её кружком. Клетка **2** *белая*.
7. **В столбце O** образовалась группа из 1 клетки, единственная в этом ряду, следовательно, весь ряд решён, отмечаем его галочкой. Оставшиеся клетки ряда – *белые*.
8. **Строка 4.** Согласно **Правилу 7**, клетки **F**, **M** принадлежат группе из 11 клеток, иначе эту группу невозможно разместить на поле. Все клетки между ними, **G-L**, *чёрные*. По **Правилу 10**, определяем, что клетка **P** – *белая*.
9. **Столбец P.** По **Правилу 9**, клетка **5** – *начальная* для группы, поэтому клетки **7** и **8** чёрные, а группа решена. Так как группа единственная в ряду, то решён и весь ряд. Отмечаем его галочкой, а оставшиеся клетки ряда помечаем как *белые*.

После этого хода на поле возникла такая забавная конфигурация клеток (Рис. 3.16).

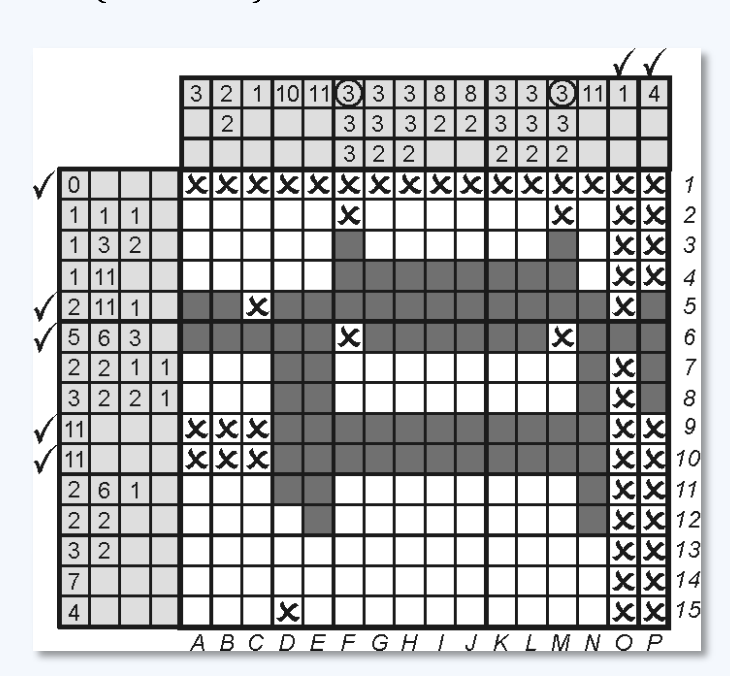


Рис. 3.16. И еще пяток

10. **В строке 7** образовались две последние группы, отмечаем их кружком, а в клетку **M** ставим крестик - она *белая*.
11. **В строке 8** вырисовалась последняя группа, а также известна конечная клетка предпоследней группы, следовательно, клетка **M** *чёрная*, а **L** – *белая*. Отмечаем обе группы кружком.
12. **В столбцах G, H, K, L** образовались первые группы из 3 клеток, отмечаем их кружком и обрамляем *белыми* клетками **3** и **4**.
13. **В строке 11** образовалась последняя группа из 1 клетки. Отмечаем её кружком, а в клетку **M** ставим *крестик*.

14. **В столбцах I, J** все чёрные клетки входят в группу из 8 клеток, поэтому неразгаданные клетки **7** и **8** между ними можно закрасить.
15. **В строке 7** получились последние две группы - ряд решён. Оставшиеся неразгаданными клетки помечаем как *белые*.
16. **Столбец С** решился сам собой – ура, то есть банзай!!!
17. **В строке 8** возникла ещё одна группа из 2 клеток, а клетка **D** – *начальная* для группы из 3 клеток, поэтому с чистой совестью мы закрашиваем клетку **F**. Со всеми группами этого ряда мы успешно справились, помечаем его *галочкой*, а оставшиеся клетки *крестиком*.

Позиция, возникшая на «доске» после 17-ого хода (следите за мыслью!) (Рис. 3.17).

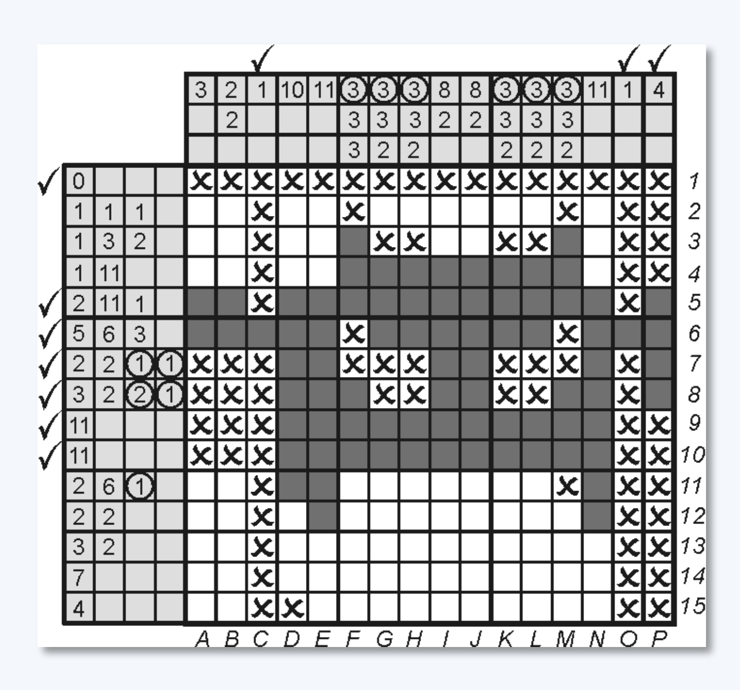


Рис. 3.17. И еще чуток

18. **В столбцах F, M** образовались вторые группы из 3 клеток, отмечаем их *кружком*. В клетку **F11** ставим *крестик*.
19. **Столбцы G, H, K, L** – клетка **9** начинает вторую группу из 3 клеток, закрашиваем клетку **11** и ставим крестик в поле **12**. Отмечаем, что группа решена. Клетка **2** может быть только *белой*, так как первая группа уже разгадана.
20. **В строке 11** образовалась группа из 2 клеток, отмечаем её *кружком*. Чёрные клетки **F, G** и **K, L** могут принадлежать только группе из 6 клеток, помечаем клетки **I, J** как *чёрные*. Группа разгадана. Разгадан полностью и весь ряд, ставим слева *галочку*. Оставшиеся клетки **A, B**, разумеется, *белые*.
21. **В столбцах I, J** образовались группы из 8 клеток, обрамляем их *крестиками*. Клетка **2** тоже *белая*. Фиксируем свои успехи.
22. **В строке 4** между белыми клетками **C** и **O** ровно 11 клеток, поэтому группа из 11 клеток решена, клетки **D, E, N** *закрашиваем*.
23. **В строке 3** клетка **M** – *начальная* для группы из 2 клеток, поэтому клетка **N** – *чёрная*. Клетка **F** – *конечная* для группы из 3 клеток, значит, клетки **D, E** также *чёрные*. Отмечаем обе группы как решённые.
24. **Столбец А.** Клетка **6** – *конечная* для группы из 3 клеток, поэтому клетка **4** *чёрная*. Группа решена, а других в этом ряду и нет. Отмечаем его *галочкой*.

25. **В столбце B** получилась одна из групп из 2 клеток, следовательно, клетка **4** - *белая*.

Ситуация на поле складывается явно в нашу пользу (Рис. 3.18).

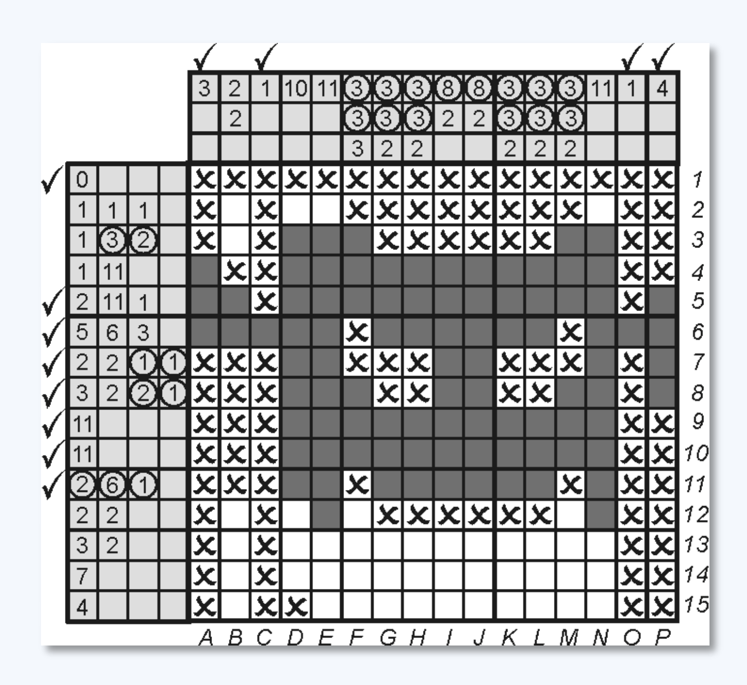


Рис. 3.18. Победа близка!

26. **В строке 4** образовалась первая группа, теперь все группы решены, а с ними и весь ряд.
27. **В строках 12, 13, 14, 15** между белыми клетками **A** и **C** не сможет разместиться ни одна группа, стало быть, клетка **B** – *белая*.
28. **В столбце B** остались только две не разгаданных клетки – как раз для первой группы. И с этим рядом покончено.
29. **В строке 2** образовалась первая группа (в клетке **B**). Последняя группа может занять только клетку **N**, так как на остальных клетках (**D, E**) не могут разместиться 2 группы.
30. **В столбце N** сама собой возникла единственная группа из 11 клеток, что позволяет нам считать клетки **13**, **14**, **15** *белыми*. И с этим рядом мы управились.
31. **В строке 12** клетка **N** - *конечная* для второй группы из 2 клеток, поэтому предыдущая клетка также *чёрная*.
32. **В столбце M** клетка *12* - начальная для группы из 2 клеток, следовательно, клетка **13** также *чёрная*. Все группы этого ряда решены, осталось поставить крестики в клетки **14** и **15**.
33. **В строке 3** образовалась группа из 1 клетки, с чем мы можем себя поздравить – теперь весь ряд решён.
34. **В строке 13** клетка **M** – *конечная* для группы из 2 клеток, значит, клетка **L** *чёрная*, а предыдущая – *белая*.
35. **В строке 14** остались неразгаданными 9 клеток (**D – L**), в которых должна разместиться группа из 7 клеток, и мы должны признать, что клетки **F-J** неизбежно *чёрные*.
36. **В столбце K** хватает места только для последней группы, и мы должны отметить свой очередной успех.
37. **В строке 14** клетка **D** *белая*, по *Правилу габаритов*.

38. **В столбце L** клетка **13** – *начальная* для группы из 2 клеток. Следующую за ней клетку мы *закрашиваем*, в последнюю ставим *крестик* – и ряд приказал долго жить.

Положение пресловутой японской головоломки критическое (Рис. 3.19).

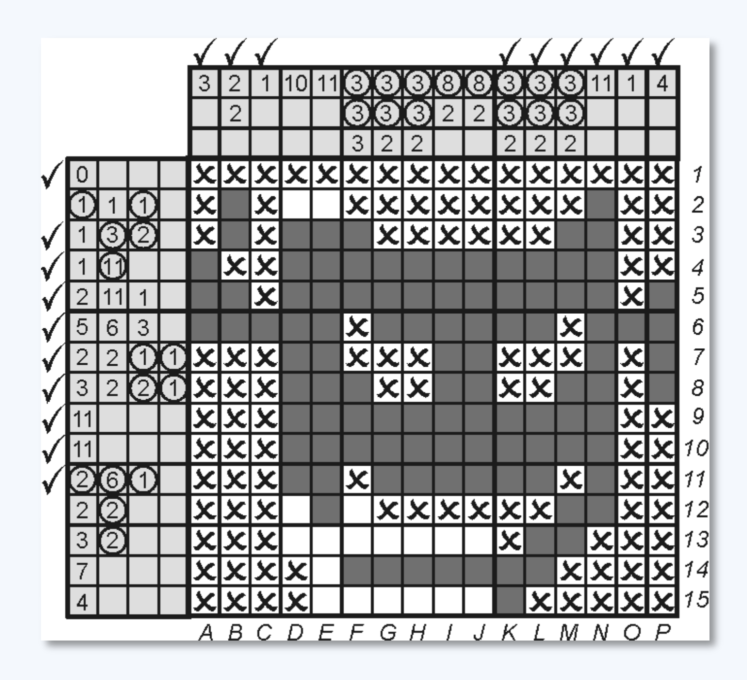


Рис. 3.19. И мы близки к победе!

39. **В строке 15** клетка **К** – *конечная* для группы из 4 клеток. И этому ряду конец.
40. **В столбцах F, G** клетка **14** – *конечная* для последней группы. Оба ряда – «туды их в качель».
41. **В строке 14** образовалась единственная группа, так что и ряду тоже *kaput*. (Не забудьте на радостях поставить ему *крестик* в клетку **E**.)
42. **В строке 12** возникла группа из 2 клеток. Клетка **D** – *белая*, а ряд благополучно *того*.
43. Со **столбцом D** разобраться проще пареной репы: клетка **13** *белая*, а клетка **2**, напротив, *чёрная*.
44. **В строке 2** образовалась последняя группа, нам осталось наградить клетку **E** *крестом* (посмертно).
45. **В столбце E** ставим жирную чёрную точку в несчастливую клетку **13**, и пусть этот ряд покоится с миром.
46. **В строке 13** получилась группа из 3 клеток. Ставим последние *кресты* в этой задаче. Аминь.

Все клетки разгаданы. Теперь, коли ещё остались силы, вы можете полюбоваться зрелыми плодами своей интеллектуальной деятельности (Рис. 3.20).

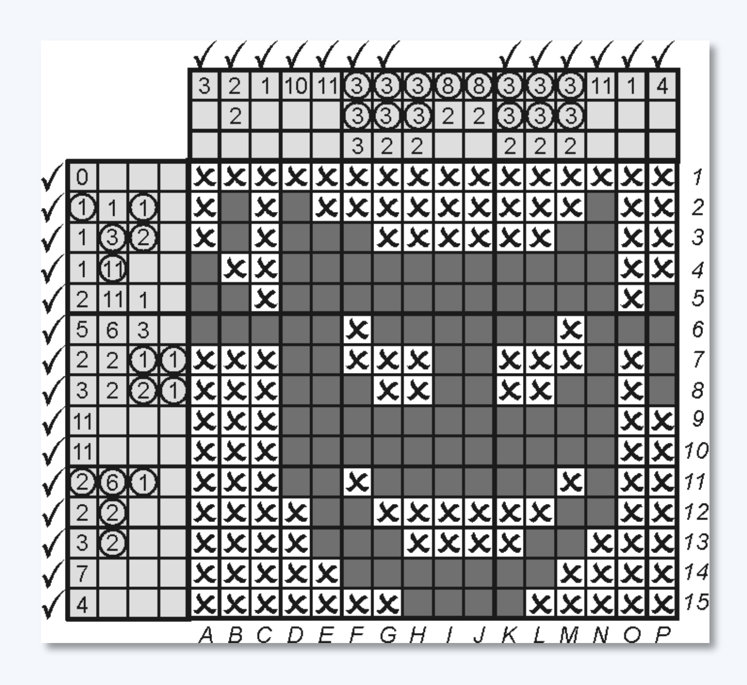


Рис. 3.20. Разделали в пух и прах!



Аналогично «раскрашиваются» и более сложные рисунки – правила решения одинаково верны для всех задач. Однако, если рисунок имеет размеры побольше нашего (а чаще всего так оно и есть), да плюс ко всему множество мелких групп, то и потрудиться придётся преизрядно. Не удивительно, что где-то можно ошибиться и вместо чёрной клетки поставить белую (или наоборот), из-за чего всё решение пойдёт дальше шиворот-навыворот и вы напрасно потратите остаток своей жизни на тщетные поиски коварных заблуждений. Более того, некоторые задачи садистского толка вообще устроены так, что их невозможно решить «чисто логически», не делая предположений насчёт цвета отдельных клеток. Скорее всего, одно из них окажется неверным, и это сильно огорчит вас, потому что придется несолоно хлебавши возвращаться назад, а, значит, что вам не избежать «протокола», в котором вы должны будете фиксировать позицию на поле перед тем, как наугад пометить клетку. Тут уж вы повеселитесь на полную катушку!

Кибернетическое убийство времени, или Компьютер-поводырь

Заграница нам поможет!

Один из посулов Остапа Бендера

Если вы любитель «рукоприкладства» в решении головоломок, но тяготитесь бумажной рутиной, то с помощью нашей будущей программы сможете решать японские головоломки на экране монитора. Хотя надо иметь ангельское терпение, чтобы часами решать на компьютере задачи вручную, когда тот же компьютер справится с ними в «уно моменто». Я надеюсь на ваше благоразумие, поэтому ручному решению японских кроссвордов в нашей программе мы уделим меньше внимания, чем автоматическому. Если же вас привлекает именно ручная, тонкая работа, то я могу порекомендовать вам несколько программ, которые помогут вам убедиться в бренности всего сущего.

1. Вас, несомненно, порадует «японским» дизайном *игра Cross Image* компании *Xdyne* (Рис. 3.21).

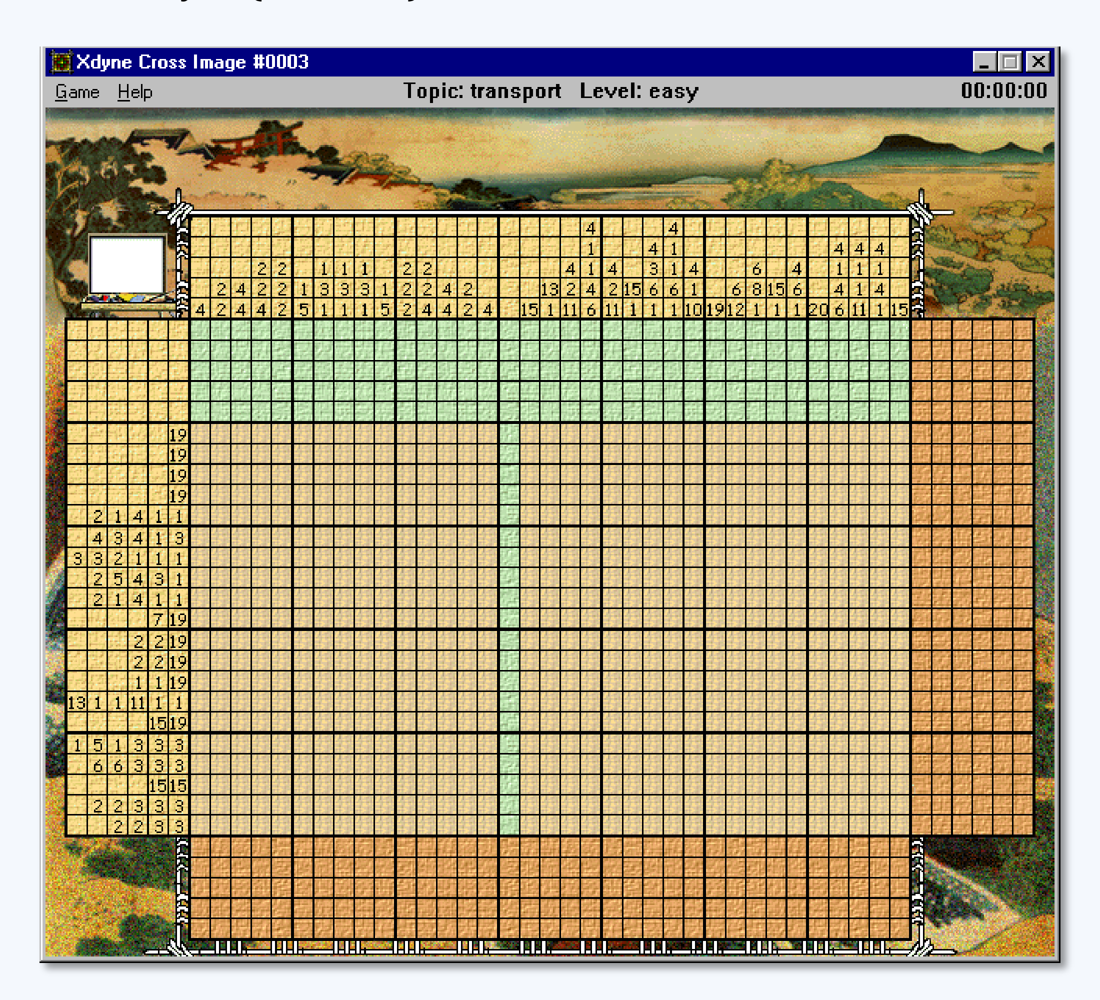


Рис. 3.21. Японисто!

Впрочем, радоваться вам придётся недолго: справившись с тремя прилагаемыми задачами, вы задумаетесь, где бы достать новые. К сожалению, ввести свои задачи вам не удастся, так что эта программа почти бесполезная.

2. Значительно «правильнее» программа Романа Гантверга *Японский кроссворд. Игра и редактор* с библиотекой из нескольких десятков заданий (Рис. 3.22, слева), к тому же вы сможете вводить и свои собственные. Для автоматического решения задач вполне уместно использовать программу *Soluter* Гантверга и Самсонова (Рис. 3.22, справа).

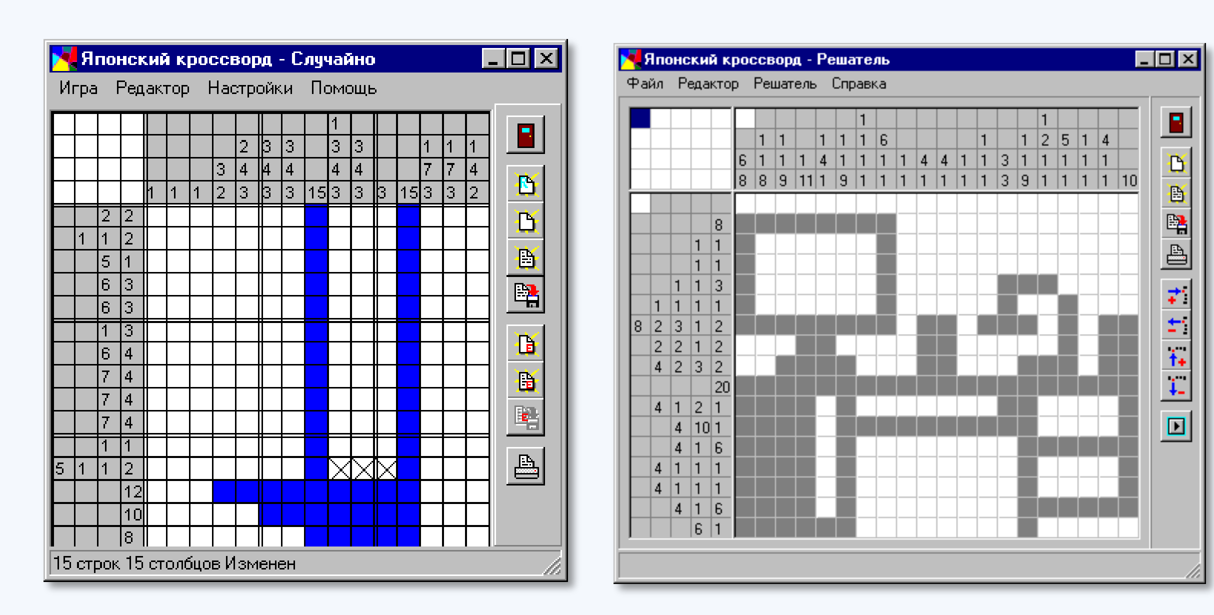


Рис. 3.22. Программы Гантверга и Самсонова

3. Не менее интересна программа *Японский кроссворд 2000* Егоркина Игоря Владимировича, которая включает 315 задач. Вы можете и самостоятельно вводить новые задания, а программа в трудную минуту придёт вам на помощь и дорешает задачу, если силы оставят вас (Рис. 3.23).

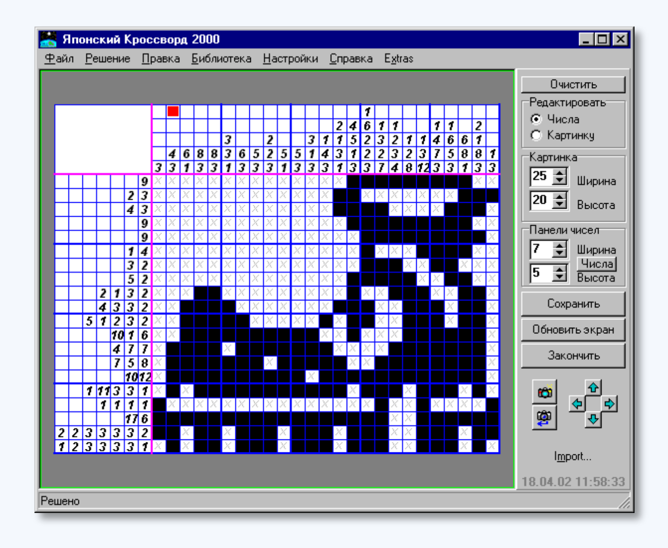


Рис. 3.23. Японский кроссворд 2000 в действии

4. А в программе Антона Головина *Японская головоломка* вы, как Фигаро или Труффальдино из Бергамо, сможете одновременно биться на нескольких фронтах, решая несколько задач в отдельном окне (Рис. 3.24).

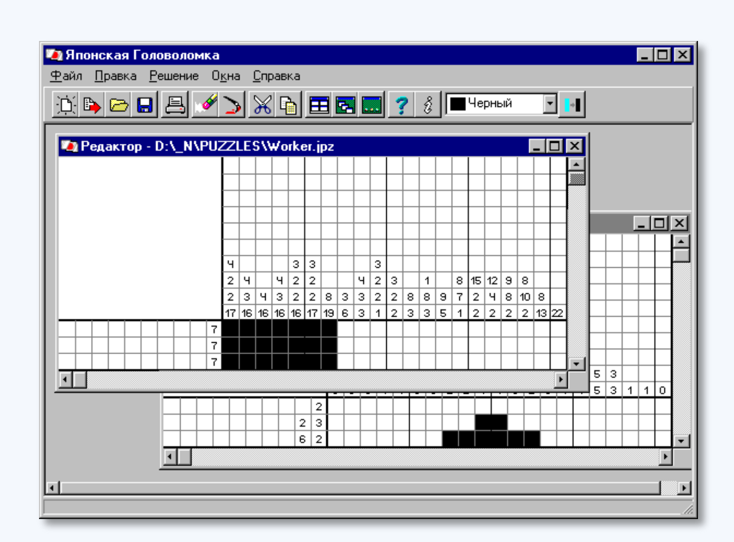


Рис. 3.24. Японская головоломка

5. Беднее по графическому исполнению, но близка по возможностям к первой программе игра *JpcWin* Антона Ильяшенко (Рис. 3.25).

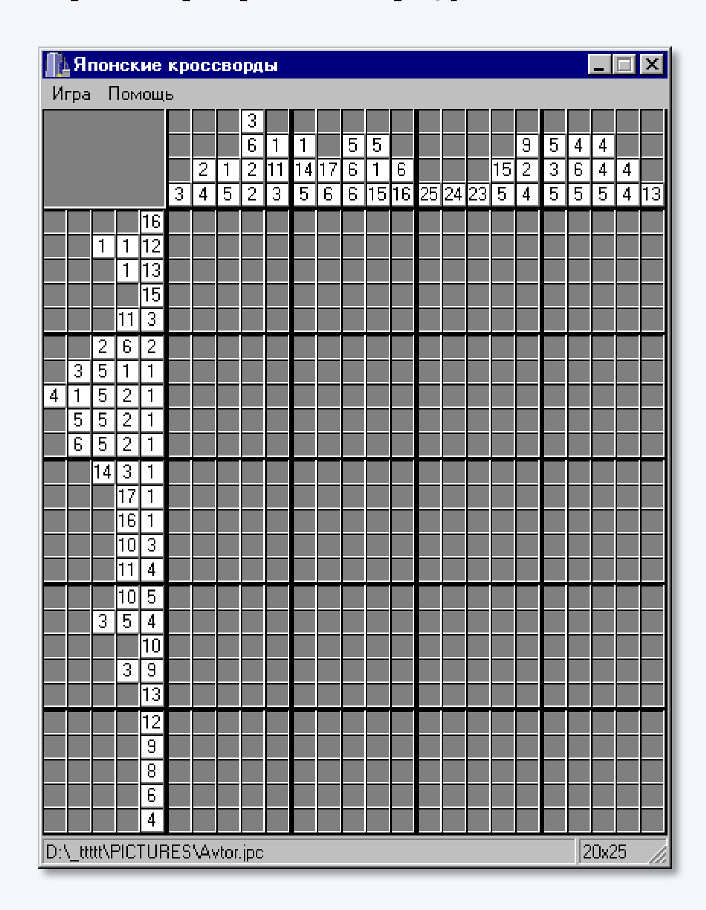


Рис. 3.25. Программа *JpcWin*

6. Очень мощная программа *Cross+A* Сергея Кутасова и Ильи Морозова умеет решать не только черно-белые (Рис. 3.26, слева), но и *цветные* задачки (Рис. 3.26, справа).

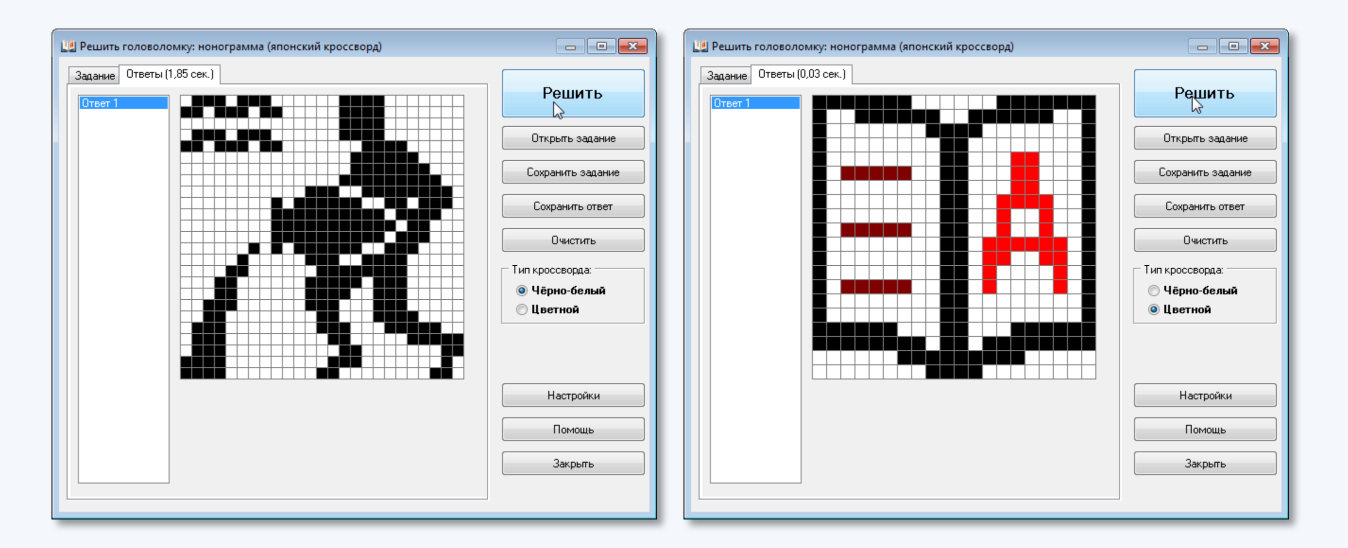


Рис. 3.26. *Cross+A* решает *Водолейную* задачу

К тому же она позволяет создавать и редактировать собственные задачи, а также поддерживает файлы разных форматов.

7. И последняя программа – *Gunn's Japanese Puzzle* – также предлагает решать только свои собственные задачи. А их прилагается не очень много…

Решительный штурм, или Как расколоть японские кроссворды

Взялся за гуж, не говори, что не дюж.

Из русского фольклора

Каждый любитель головоломок погорячее, как и былинный витязь на распутье, должен выбрать одну из дорог: решать задачу «вручную» или с помощью компьютера. Оба пути не из лёгких (бытует заблуждение, что на компьютере задачу и дурак решит), а потому по-своему интересны (а зачем вообще браться за простые задачи, например, сканворды?). Но тот, кто предпочтёт «сотрудничество» с компьютером, получит двойное удовольствие – и задачу решит (и не только одну, но и все подобные), и программу напишет, что само по себе во много раз интереснее любой головоломки. Когда *ваша* программа начнёт щёлкать сложнющие задания как орешки, вас охватит ни с чем несравнимая радость творца, из «ничего» создавшего целый мир, в котором всё движется и шевелится, покоряясь его воле.

Мы напишем программу, которой будет по зубам любой японский кроссворд. Она может послужить основой для вашего творчества – изучайте, изменяйте и улучшайте её – до тех пор, пока не удовлетворитесь полученным результатом.

А начнём мы наш новый проект по привычке с соблюдения формальностей, то есть создадим форму солидных размеров, так как многие задачи имеют до 50-80 клеток в ширину и высоту. Свойства формы:


```
Width=729
Height=576
BorderIcons= [biSystemMenu,biMinimize]
BorderStyle=bsSingle
Caption=' ЯПОНСКИЙ РИСУНОК'
Position=poDesktopCenter
```

По образу и подобию «бумажных» задач мы разместим на форме 3 сетки, роль которых исполнят фавориты наших программ – компоненты *TDrawGrid*.

dgPole (игровое поле), его свойства:

```
ColCout=50
DefaultColWidth=14 (все ячейки будут иметь размер 14 x 14 пиксе-
лей, иначе цифры будут слишком мелкими и плохо различимыми на
экране)
DefaulDrawig=False (выводить содержимое ячеек будем сами)
DefaultRowHeight=14
FixedCols=0
FixedRows=0
Height=419
Left=40
Options=[goVertLine,goHorzLine] (оставим только вертикальные и го-
ризонтальные линии, другие возможности компонента нам не понадо-
бятся)
RowCout=50
ScrollBars=ssNone
Top=76
Width=449
```

dgColsNum (верхнее числовое поле) со свойствами:

```
ColCout=10
DefaultColWidth=14
DefaulDrawig=False
DefaultRowHeight=14
FixedCols=0
FixedRows=0
Height=33
Left=44
```

```
Options=[goVertLine,goHorzLine]
RowCout=4
ScrollBars=ssNone
Top=76
Width=449
```

dgRowsNum (левое числовое поле):

```
ColCout=4
DefaultColWidth=14
DefaulDrawig=False
DefaultRowHeight=14
FixedCols=0
FixedRows=0
Height= 421
Left=4
Options=[goVertLine,goHorzLine]
RowCout=10
ScrollBars=ssNone
Top=76
Width=33
```


Так как размеры полей могут превышать возможности формы, то нам придётся позаботиться об их *прокрутке*, чтобы добраться и до скрытых от нашего пытливого взора частей рисунка. Для воплощения этой мечты удобнее всего воспользоваться услугами компонента *TScrollBar* TScrollBar со страницы *Standard Панели компонентов*. Нетрудно догадаться, что их потребуется ровно два – для вертикальной - *sbVert* - и горизонтальной - *sbHorz* – прокрутки. Большинство их свойств можно не изменять в *Инспекторе объектов*, так как полосы прокрутки нам придётся устанавливать на форме в соответствии с размерами полей. Но о некоторых, постоянных свойствах лучше позаботиться сразу. Для *sbVert* установите:

```
Cursor= crHandPoint (точно такой же курсор устанавливайте и для
других компонентов, на которые придётся «нажимать»)
Kind= sbVertical
LargeChange=5
PageSize=5
```

Для *sbHorz*:

```
Kind= sbHorizontal
LargeChange=5
PageSize=5
```

Для вывода актуальной информации мы установим на форме компонент *StatusBar1* TStatusBar :

```
Align=alBottom
```

```
Autohint=True
Height=25
Showhint=True
```

Дважды щёлкните на нём и в *Редакторе* разметьте панели:

```
 0 (в крайней левой панельке будут выводиться подсказки (hint):
Alignment=taLeftJustify
Width=196;
 1 (следующая панелька покажет размеры загруженной картинки):
Alignment=taCenter
Text=' 28 x 28'
Width=60;
 2 (далее следует информация о числе клеток в картинке):
Alignment=taCenter
Text=' Клеток: 0'
Width=90;
 3 (горизонтальная координата курсора на игровом поле):
Alignment=taCenter
Text=' X= 0'
Width=40;
 4 (вертикальная координата курсора на игровом поле):
Alignment=taCenter
Text=' Y= 0'
Width=40;
 5 (режим работы программы: ожидает ли она наших действий или ре-
шает задачу):
Alignment=taCenter
Text=' ОЖИДАНИЕ'
Width=90;
 6 (можете приспособить для своих нужд).
```

Под заголовком формы разместите кнопки *TSpeedButton*:

sbtNewFig – служит для изменения размеров полей:

```
Height=36
Width=36 (все остальные кнопки точно такие же квадратные)
Hint=' Размеры|Изменить размеры полей'
Left=4
Showhint=True (установите у всех кнопок)
Top=0 (тоже у всех кнопок));
```

sbtClearPole – очищает все поля, игровое поле становится серым, верхнее и левое – белыми, без чисел:

```
Hint=' Очистить| Очистить поля';
```

sbtWhiteGrid – очищает все поля, игровое поле становится белым, что удобно для рисования новых картинок:

```
Hint=' Белая сетка|Белая сетка для рисования';
```

sbtNumbers – оцифровывает нарисованную вами картинку:

```
Caption='№'
Hint=' Оцифровать|Оцифровать рисунок';
```


sbtLoadFig – загружает сохранённые ранее картинки:

```
Hint=' Загрузить | Загрузить рисунок'
```

Для её работы необходим компонент *TOpenDialog* со страницы *Dialogs Панели компонентов*;

sbtOpenPicture – загружает растровые рисунки и значки для создания из них новых рисунков:

```
Hint=' Импорт|Импорт растров и значков'
```

Для её работы установите компонент *TOpenPictureDialog* TOpenPictureDialog с той же страницы;

sbtSaveFig – сохраняет на диске задачу (данные из числовых полей):

```
Hint=' Записать|Записать задачу')
```

Потребуется компонент *TSaveDialog* TSaveDialog, который вы найдёте там же;

sbtSavePic – сохраняет на диске рисунок (из игрового поля):

```
Hint=' Записать рисунок|Записать рисунок';
```

sbtStart – начинает автоматическое решение задачи:

```
Hint=' Решить|Решить задачу';
```

sbtStop STOP – останавливает решение задачи, если оно затягивается:

```
Hint= ' Остановить|Остановить поиск';
```

sbtExit – заканчивает работу с программой:

```
Hint=' Выход|Выход из программы';
```

Ещё три кнопки служат *переключателями*.

sbtSound – включает и выключает звуковые эффекты:

```
AllowAllUp=True
GroupIndex=1
Hint=' Звук|Звук вкл-откл';
```


sbtMove – включает и выключает режим перемещения рисунка:

```
AllowAllUp=True
GroupIndex=2
Hint=' Двигать|Двигать рисунок';
```

sbtDraw – включает и выключает режим рисования картинки и ручного решения задачи:

```
AllowAllUp=True
GroupIndex=3
Hint=' Рисование, решение|Рисовать картинку, решать задачу'.
```

И последнее, что нам осталось, - уютно разместить на форме две *метки* (вы можете легко обойтись и без них):

lblReady – показывает число разгаданных клеток рисунка (при автоматическом решении):

```
Hint=' Разгадано|Число разгаданных клеток';
```

lblLevel – показывает сложность решения задачи (число предположений, «сомнительных» клеток:

```
Hint=' Уровень|Сложность решения задачи'.
```

В итоге перед вашим взором должна открыться такая картина (Рис. 3.27).

Вы можете переставить «мебель» по своему вкусу, да только зачем? Сейчас для нас куда важнее наполнить форму нужным содержанием, чтобы она не только радовала взор, но и славно на нас работала.

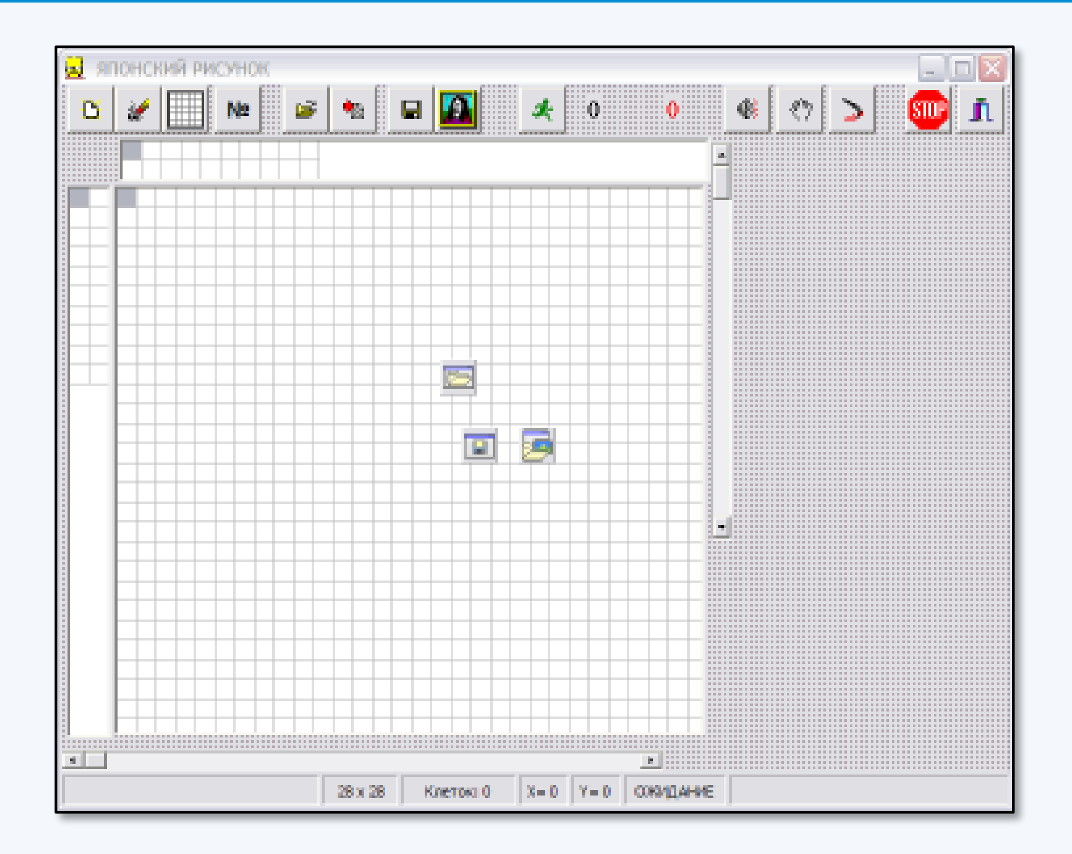


Рис. 3.27. Лицевая сторона нашего приложения

Занимательная механика, или За кулисами формы

Природа не терпит пустоты.

Сказал один мудрец

Но часто наполняет её всякой дрянью.

Добавил другой

Предположим, что вас сильно огорчила некая японская задача, которая спутала вам все карты и жизненные планы, а вы не такой человек, чтобы безропотно сносить удары судьбы. Тогда вооружитесь компьютером и нанесите ответный удар.

Итак, ваш противник в виде несгибаемой задачи строит вам фигуры со станицы газеты, а то и журнала. Компьютер газет не читает (да и журналов тоже), поэтому нужно подать ему головоломку на *блюдечке с голубой каёмочкой*. Сервировать блюдо можно по-разному. Первое что приходит в голову – просто перенести числа из газеты в числовые поля нашей программы.

Компонент *TDrawGrid* вполне уместен для отображения информации, но хранить данные он не умеет, поэтому каждой сетке мы сопоставим *массив*. Перед определением типа формы запишем (все остальные переменные, не относящиеся к процедурам, мы также будем объявлять в этом разделе):

```
var
  //массив, в котором хранится копия поля
  masPole : TPole;
```

Так как подобные массивы пригодятся нам и в других случаях, то перед разделом переменных мы опишем *тип игрового поля*:


```
//данные игрового поля
type  TPole  =  array[0..MAX_POLE_WIDTH-1,  0..MAX_POLE_HEIGHT-1]of
Integer;
```

Верхние границы массива лучше задать в виде *констант* – и понятнее, и легче изменить в случае необходимости. Константы объявляем сразу после раздела *uses*:

```
unit NipponUnit;

///////////////////////////////////////////////////////////////////////////
//                                                                       //
//             ПРОГРАММА ДЛЯ РЕШЕНИЯ ЯПОНСКИХ ГОЛОВОЛОМОК               //
//                                                                       //
///////////////////////////////////////////////////////////////////////////

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  ExtCtrls, StdCtrls, Grids, ComCtrls, Buttons, mmsystem, ExtDlgs;

const
  NAME_PROG = 'ЯПОНСКИЙ РИСУНОК';
  MAX_POLE_WIDTH = 70;    //макс. ширина поля в клетках
  MAX_POLE_HEIGHT = 50;   //макс. высота поля в клетках
```

Установленные предельные размеры сетки позволяют загрузить практически все существующие задачи. Если же вам этого недостаточно, вы можете увеличить их. Но имейте в виду, что большие и плохо составленные задачи могут потребовать достаточно много времени для решения!

В массиве игрового поля мы будем хранить данные о *цвете* клеток. Помните, как нам приходилось хитрить, чтобы отметить разгаданные белые клетки? И это всё из-за того, что не решённые ещё клетки на бумаге тоже белые. Но компьютер не газета, поэтому мы просто сделаем *неиз-*

вестные клетки *серыми*, а *известные*, как и полагается, *белыми* и *чёрными*. Для удобства пользования и в целях профилактики склероза объявим соответствующие *константы*:

```
  NUM_WHITE = 2;         //белая клетка в masPole
  NUM_BLACK = 1;         //чёрная клетка в masPole
  NUM_GRAY = 0;          // серая клетка в masPole (фон)
```

Эти же клетки на игровом поле мы будем окрашивать в нужные цвета:


```
  COLOR_WHITE: TColor= $FFFFFF;  //белая клетка на поле
  COLOR_BLACK: TColor= $0;       //чёрная клетка на поле
  COLOR_GRAY: TColor= $7F7F7F;   //серая клетка на поле
```

Текущее значение размеров поля мы будем хранить в переменных:

```
  POLE_WIDTH: integer = 28;     //ширина поля в клетках
  POLE_HEIGHT: integer = 28;    //высота поля в клетках
```

Массивы для числовых полей отличаются от массива игрового поля не только размерами, но и структурой:

```
  //массив, в котором хранятся верхние числа:
  masColsNum   :   array[0..MAX_POLE_WIDTH-1,   -1..MAX_CELLNUM-1]of
TCell;
  //массив, в котором хранятся левые числа:
  masRowsNum   :  array[-1..MAX_CELLNUM-1,   0..MAX_POLE_HEIGHT-1]of
TCell;
```

В каждой ячейке массивов будет храниться *число*, записанное в соответствующей клетке сетки (нам понадобится не только само число, но и его представление в виде *строки*), а также *статус группы*, для описания которого мы введём новый *тип*:

```
//статус групп в числовых полях:
//stWhite - группа не решена,
//stYellow - группа решена,
//stGreen - ряд решён
type TStatusGroup=(stWhite, stYellow, stGreen);
//данные для каждой клетки числовых полей:
type TCell = Record
  sNum: String;            //строковое представление числа в клетке
  Num: integer;               //число в клетке
  StatusGroup: TStatusGroup; //статус группы
end;
```

Кроме того, в ячейках с индексом –1 мы будем хранить *число групп* в данном ряду.

Объявим также *константу*, ограничивающую размеры числовых полей:

```
MAX_CELLNUM = 14;        //макс. ширина/высота числовых полей
```

Этого значения вполне достаточно для большинства заданий (правильнее сказать, для всех *разумных* заданий). Впрочем, никто не помешает вам увеличить его по собственному разумению. О последствиях этого читайте выше.

Переменные для текущих (установленных) размеров числовых полей:


```
  LEFT_POLE_WIDTH: integer = 2; //ширина левого числового поля в
клетках
  TOP_POLE_HEIGHT: integer = 2; //высота верхнего числового поля в
клетках
```

Очевидно, что высота *левого* числового поля равна *высоте* игрового поля, а *ширина* верхнего числового поля – *ширине* игрового поля (если для вас это не очевидно, а невероятно, то просто внимательно посмотрите на картинку с формой).

Так как размеры экрана обычного монитора не позволяют целиком отобразить всё поле больших размеров, то мы должны ввести *константы* для предельных размеров «вырезки» поля:

```
  MAX_ALL_CELLS_HEIGHT = 30;     //макс. высота игрового и числово-
го полей
                                //(видимая часть)
  MAX_ALL_CELLS_WIDTH = 46;      //макс. ширина игрового и числово-
го полей
```

Для пущего куражу введём несколько нестандартных *курсоров* (в разделе констант):

```
  //курсоры:
  crHand : integer= 4;
  crMove : integer= 5;
  crKistj : integer= 7;
```

Не забудьте подключить к проекту *файл ресурсов* со своими курсорами:

```
{$R *.DFM}
{$R MyCursors.res}
```

За *режим работы* программы будет отвечать переменная:

```
 status: string='';
```

Благодаря ей, нажимая любую кнопку (я надеюсь, вы понимаете, о каких кнопках речь), мы сможем узнать, что делает наша программа. Если она занята решением задачи, то беспокоить её не следует. Можно отключать все «лишние» кнопки на время решения задачи, но это потребует б**о**льших усилий.

Название картинки мы будем хранить в *переменной*:

```
  NameFig: string= 'temp';
```


Если картинка не загружалась и не сохранялась на диске, то имя ей - *temp*. Название картинки, чтобы не забыть оного, мы будем выводить в *заголовке* формы.

Чтобы лучше ориентироваться в «пространстве», мы будем следить за положением курсора в сетках:

```
  //координаты клетки с курсором:
  cellMouse: TPoint;      //игровое поле
  cellMouseTop: TPoint;   //верхнее числовое поле
  cellMouseLeft: TPoint; //левое числовое поле
```

Ещё две переменные будут следить за *количеством* клеток:

```
  AllCells: Integer=0;    //всего клеток на поле
  ReadyCells: Integer=0; //разгадано клеток
```

Теперь у нас есть всё для того, чтобы начать творить. Объявим две *процедуры*, с помощью которых можно вернуть массивам девственную чистоту:

```
    { Private declarations }
    procedure Clear_masColsNum;
    procedure Clear_masRowsNum;
```

Сами процедуры очень просты: мы стираем все числа в ячейках и указываем, что ряд не решён:

```
//ОЧИСТИТЬ МАССИВ ВЕРХНЕГО ЧИСЛОВОГО ПОЛЯ
procedure TForm1.Clear_masColsNum;
var
  i,j: integer;
begin
  For i := 0 To MAX_POLE_WIDTH-1 do
    for j := -1 to  MAX_CELLNUM - 1 do
    begin
      masColsNum[i,j].sNum:='';              // - числа нет
      masColsNum[i,j].StatusGroup:= stWhite;// - ряд не решён
    end
```

```
end;
//ОЧИСТИТЬ МАССИВ ЛЕВОГО ЧИСЛОВОГО ПОЛЯ
procedure TForm1.Clear_masRowsNum;
var
  i,j: integer;
begin
  For i := -1 To MAX_CELLNUM-1 do
    for j := 0 to  MAX_POLE_HEIGHT - 1 do
    begin
      masRowsNum[i,j].sNum:='';             // - числа нет
      masRowsNum[i,j].StatusGroup:= stWhite;// - ряд не решён
    end
end;
```


После этих процедур объявляем ещё одну:

```
    procedure InvalidateGrids;
```

Она будет «освежать» игровое и числовые поля после глобальных изменений:

```
//ОБНОВИТЬ ПОЛЯ
procedure TForm1.InvalidateGrids;
begin
  dgPole.Invalidate;
  dgColsNum.Invalidate;
  dgRowsNum.Invalidate;
end;
```

В типе формы объявите процедуру, которая будет *подготавливать* сетки заданных размеров и при необходимости добавлять полосы прокрутки:

```
procedure  Prepare(ColCount,  RowCount,  TopCount,  LeftCount:  Inte-
ger);
```

```
//ПОДГОТОВИТЬ НОВУЮ ИГРУ
procedure TForm1.Prepare(ColCount, RowCount, TopCount, LeftCount:
Integer);
//ColCount - число колонок на игровом поле
//RowCount - число строк на игровом поле
//TopCount - число строк на верхнем поле
//LeftCount - число колонок на левом поле
var
  w,h,lw: integer;
  i,j,n: integer;
begin
  //скорректировать размеры полей:
  if ColCount< 2 then ColCount:= 2;
  if RowCount< 2 then RowCount:= 2;
  if TopCount< 1 then TopCount:= 1;
  if LeftCount< 1 then LeftCount:= 1;
```

```
  //
  //============= игровое поле ===============
  //
  //размер клетки в пикселах:
  w:= dgPole.DefaultColWidth;
  h:= dgPole.DefaultRowHeight;
  //толщина линий:
  lw:= dgPole.GridLineWidth;
  //размеры игрового поля в клетках:
  dgPole.ColCount:= ColCount;
  dgPole.RowCount:= RowCount;
  i:= ColCount;
  j:= RowCount;
  //если ширина игрового поля + ширина левого числового поля
  //превышают MAX_ALL_CELLS_WIDTH клеток,
  //то поставить вертикальную полосу прокрутки:
  if ColCount > MAX_ALL_CELLS_WIDTH  - LeftCount then begin
    i:= MAX_ALL_CELLS_WIDTH  - LeftCount;
    sbHorz.Visible:= true
    end
  else sbHorz.Visible:= false;
  //если высота игрового поля + высота верхнего числового поля
  //превышают MAX_ALL_CELLS_HEIGHT клеток,
  //то поставить горизонтальную полосу прокрутки:
  if RowCount > MAX_ALL_CELLS_HEIGHT  - TopCount then begin
    j:= MAX_ALL_CELLS_HEIGHT  - TopCount;
    sbVert.Visible:= true
    end
  else sbVert.Visible:= false;
  //размеры в пикселах видимой части игрового поля:
  dgPole.Width:= 3 + (w + lw)* i+1;
  dgPole.Height:= 3 + (h + lw)* j+1;
  //
  //==================== числовые поля =========================
  //
  //размеры клеток числовых полей = размерам клеток игрового поля:
  dgColsNum.DefaultColWidth:= w;
  dgColsNum.DefaultRowHeight:= h;
  dgRowsNum.DefaultColWidth:= w;
  dgRowsNum.DefaultRowHeight:= h;
  //размеры числовых полей:
  dgColsNum.ColCount:= ColCount; // = ширине игрового поля
  dgColsNum.Width:= dgPole.Width;
  dgColsNum.RowCount:= TopCount; // задаётся
  //высота верхнего числового поля в пикселах:
  dgColsNum.Height:= 3 + (h + lw)* TopCount;
   //высота левого числового поля в клетках = высоте игрового по-
ля:
  dgRowsNum.RowCount:= dgPole.RowCount;
  //высота левого числового поля в пикселах:
  dgRowsNum.Height:= dgPole.Height;
  dgRowsNum.ColCount:= LeftCount; // задаётся
  //ширина левого числового поля в пикселах::
```

```
  dgRowsNum.Width:=3 + (h + lw)* LeftCount;
  //положение на форме:
  dgColsNum.left:= dgRowsNum.Left + dgRowsNum.Width;
  dgRowsNum.top:=  dgColsNum.top +  dgColsNum.Height;
  dgPole.top:= dgRowsNum.top;
  dgPole.Left:=dgColsNum.Left;
  //
  //==============  полосы прокрутки  =================
  //
  //разместить полосы прокрутки рядом с игровым полем:
  if sbVert.visible then
  begin
    //коорд. левой стороны верт. полосы прокрутки:
    sbVert.Left:= dgPole.Left+ dgPole.Width+ 5;
    //её высота:
    sbVert.Height:= dgPole.Height;
    //коорд. верхней стороны:
    sbVert.Top := dgPole.top;
    //макс. позиция:
    n:= dgPole.RowCount - (MAX_ALL_CELLS_HEIGHT-
dgColsNum.RowCount);
    if n< sbVert.LargeChange then n:= sbVert.LargeChange;
    sbVert.Max := n+4;
  end;
  if sbHorz.visible then
  begin
    //коорд. верхней стороны гориз. полосы прокрутки:
    sbHorz.Top := dgPole.top+ dgPole.Height+ 5;
    //ширина:
    sbHorz.Width:= dgPole.Width;
    //коорд. левой стороны:
    sbHorz.Left:= dgPole.Left;
    //макс. позиция:
    n:= dgPole.ColCount - (MAX_ALL_CELLS_WIDTH-
dgRowsNum.ColCount);
    if n< sbHorz.LargeChange then n:= sbHorz.LargeChange;
    sbHorz.Max:= n;
  end;
  //вывести размеры фигуры в панели:
  statusbar1.Panels[1].text:= inttostr(ColCount)+ ' x ' + intto-
str(RowCount);
  //подсчитать число клеток в фигуре:
  AllCells:= ColCount * RowCount;
  statusbar1.Panels[2].text:='Клеток: '+inttostr(AllCells);
end; //Prepare
```

Если картинка не видна на экране целиком, то её можно перемещать с помощью полос прокрутки. Хитрость же заключается в том, что при сдвиге *игрового* поля по горизонтали одновременно должно сдвигаться и *верхнее* числовое поле, а при сдвиге по вертикали – *левое* числовое поле, иначе данные в них сместятся по отношению к рядам игрового поля.

```
//СИНХРОННО СДВИГАТЬ ПОЛЯ ПО ГОРИЗОНТАЛИ
procedure TForm1.sbHorzScroll(Sender: TObject; ScrollCode: TScrollCode;
  var ScrollPos: Integer);
begin
  dgPole.LeftCol:= ScrollPos; dgColsNum.LeftCol:= dgPole.LeftCol;
end;
//СИНХРОННО СДВИГАТЬ ПОЛЯ ПО ВЕРТИКАЛИ
procedure TForm1.sbVertScroll(Sender: TObject; ScrollCode: TScrollCode;
  var ScrollPos: Integer);
begin
  dgPole.TopRow := ScrollPos; dgRowsNum.TopRow:= dgPole.TopRow;
end;
```


Щёлкните по носу кнопку *sbtClearPole* и напишите:

```
//ОЧИСТИТЬ ПОЛЯ
procedure TForm1.sbtClearPoleClick(Sender: TObject);
var
  i,j: integer;
begin
  if status='ПОИСК' then exit;
  //все клетки поля - серые:
  For i := 0 To POLE_WIDTH-1 do
    for j := 0 to  POLE_HEIGHT - 1 do
      masPole[i,j]:= NUM_GRAY;
  //очистить цифровые поля:
  Clear_masColsNum;
  Clear_masRowsNum;
  InvalidateGrids;
  //фигура не загружена:
  NameFig:='temp';
  //вывести в заголовке формы имя временного файла:
  caption:= NAME_PROG + '   [' + NameFig + ']';
  //координаты мыши:
  cellmouse.x:=-1; cellmouse.y:=-1;
end;
```

Нажимайте на неё всякий раз, когда вам потребуется очистить *все* поля.

Аналогично поступите и с кнопкой *sbtWhiteGrid*:

```
//ОЧИСТИТЬ ПОЛЕ ДЛЯ РИСОВАНИЯ
procedure TForm1.sbtWhiteGridClick(Sender: TObject);
var
  i,j: integer;
begin
  if status='ПОИСК' then exit;
  //все клетки - белые:
  dgPole.Canvas.Brush.Color:=COLOR_WHITE;
  For i := 0 To MAX_POLE_WIDTH-1 do
```

```
    for j := 0 to  MAX_POLE_HEIGHT - 1 do
      masPole[i,j]:= NUM_WHITE;
  //очистить цифровые поля:
  Clear_masColsNum;
  Clear_masRowsNum;
  InvalidateGrids;
  //фигура не загружена:
  NameFig:='temp1';
  //вывести в заголовке формы имя загруженного файла:
  caption:= NAME_PROG + '   [' + NameFig + ']';
  //координаты мыши:
  cellmouse.x:=-1; cellmouse.y:=-1;
end;
```


Она пригодится нам в скором светлом будущем, когда мы начнём *рисовать* свои картинки на игровом поле. Обе эти процедуры различаются только одной строкой, поэтому правильнее будет объединить их в одну (домашнее задание!).

Все данные о группах и цвете клеток на игровом поле хранятся в массивах, и в сетках на форме ничего не появится, пока мы сами не позаботимся об этом. Как обычно, прорисовка клеток всех сеток происходит при вызове их метода *DrawCell*.

Для *игрового* поля это делается так. Определяем, какое число стоит в массиве на соответствующем месте, и закрашиваем клетку нужным цветом. Затем для лучшей ориентировки на поле проводим *красные линии* (если они на вас действуют, как красная тряпка на быка, просто сотрите часть кода).

```
//НАРИСОВАТЬ КЛЕТКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var  area: tRect;
begin
  //закрасить клетку нужным цветом:
  area:=dgPole.CellRect(ACol, ARow);
  case masPole[ACol, ARow] of
    NUM_WHITE: dgPole.Canvas.Brush.Color:=COLOR_WHITE;
    NUM_BLACK: dgPole.Canvas.Brush.Color:=COLOR_BLACK;
    NUM_GRAY: dgPole.Canvas.Brush.Color:=COLOR_GRAY;
  end;
  dgPole.Canvas.FillRect(area);
  //проводим красные линии через 5 клеток:
  if (ACol>0) and (ACol mod 5 = 0) then begin
    dgPole.Canvas.Pen.Color:= clRed;
    dgPole.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgPole.Canvas.LineTo(Rect.Left,Rect.Bottom)
```

```
  end;
  if (ARow>0) and (ARow mod 5 = 0) then begin
    dgPole.Canvas.Pen.Color:= clRed;
    dgPole.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgPole.Canvas.LineTo(Rect.Right,Rect.Top)
  end;
end; //dgPoleDrawCell
```

В *числовых* полях мы действуем аналогично, но нам нужно ещё вывести стоящие в клетках *числа*:


```
//ПРОРИСОВАТЬ ЯЧЕЙКУ
procedure TForm1.dgColsNumDrawCell(Sender: TObject; ACol, ARow:
Integer;
  Rect: TRect; State: TGridDrawState);
var
  sNum: string;
begin
  //если столбец решён, отмечаем его зелёным цветом:
  case masColsNum[ACol,-1].StatusGroup of
    stGreen: dgColsNum.Canvas.Brush.Color:= RGB(0,255,0);
    else begin  //- столбец не решён
      //если группа решена, выделить её жёлтым цветом:
      if masColsNum[ACol,ARow].StatusGroup= stYellow then
        dgColsNum.Canvas.Brush.Color:= clYellow
      else dgColsNum.Canvas.Brush.Color:= clWhite; //- иначе белым
    end
  end;
  dgColsNum.Canvas.FillRect(Rect);
  //выводим число в клетке (ACol, ARow):
  sNum:= masColsNum[ACol, ARow].sNum;
  with Rect, dgColsNum.Canvas do
    textrect(Rect, left+(right-left-textwidth(sNum)) div 2,
             top+(bottom-top-textheight(sNum)) div 2, sNum);
  //нарисовать красные линии через 5 клеток:
  if (ACol>0) and (ACol mod 5 = 0) then begin
    dgColsNum.Canvas.Pen.Color:= clRed;
    dgColsNum.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgColsNum.Canvas.LineTo(Rect.Left,Rect.Bottom)
  end;
  if (ARow>0) and (ARow mod 5 = 0) then begin
    dgColsNum.Canvas.Pen.Color:= clRed;
    dgColsNum.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgColsNum.Canvas.LineTo(Rect.Right,Rect.Top)
  end;
end; // dgColsNumDrawCell

//ПРОРИСОВАТЬ ЯЧЕЙКУ
procedure TForm1.dgRowsNumDrawCell(Sender: TObject; ACol, ARow:
Integer; Rect: TRect;
  State: TGridDrawState);
var
```

```
  sNum: string;
begin
  //если столбец решён, отмечаем его зелёным цветом:
  case masRowsNum[-1,ARow].StatusGroup of
    stGreen: dgRowsNum.Canvas.Brush.Color:= RGB(0,255,0);
    else begin  //- столбец не решён
      //если группа решена, выделить её жёлтым цветом:
      if masRowsNum[ACol,ARow].StatusGroup= stYellow then
        dgRowsNum.Canvas.Brush.Color:= clYellow
      else dgRowsNum.Canvas.Brush.Color:= clWhite; //- иначе белым
    end
  end;
  dgRowsNum.Canvas.FillRect(Rect);
  sNum:= masRowsNum[ACol, ARow].sNum;
    with Rect, dgRowsNum.Canvas do
    textrect(Rect, left+(right-left-textwidth(sNum)) div 2,
             top+(bottom-top-textheight(sNum)) div 2, sNum);
  //нарисовать красные линии через 5 клеток:
  if (ACol>0) and (ACol mod 5 = 0) then begin
    dgRowsNum.Canvas.Pen.Color:= clRed;
    dgRowsNum.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgRowsNum.Canvas.LineTo(Rect.Left,Rect.Bottom)
  end;
  if (ARow>0) and (ARow mod 5 = 0) then begin
    dgRowsNum.Canvas.Pen.Color:= clRed;
    dgRowsNum.Canvas.MoveTo(Rect.Left,Rect.Top);
    dgRowsNum.Canvas.LineTo(Rect.Right,Rect.Top)
  end;
end; // dgRowsNumDrawCell
```


Сделайте двойной щелчок на форме и в открывшейся заготовке процедуры напишите:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //очистить поля:
  sbtClearPoleClick(self);
  //вывести сетки заданных размеров:
  Prepare(POLE_WIDTH,POLE_HEIGHT,                  TOP_POLE_HEIGHT,
LEFT_POLE_WIDTH);
  //загрузить курсоры:
  Screen.Cursors[crKistj] := LoadCursor(hinstance,'kistj');
  Screen.Cursors[crHand] := LoadCursor(hinstance,'hand');
  Screen.Cursors[crMove] := LoadCursor(hinstance,'move');
  //установить курсор для игрового поля:
  dgPole.Cursor :=crHand;
end;
```

Осталось нагрузить делом кнопку *sbtExit*:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close
end;
```

- и можно, наконец, запустить программу.

Работать она будет исправно: все поля и кнопки на месте и ждут ваших решительных действий. Но, скорее всего, нужные нам размеры полей не совпадут с теми, что установлены по умолчанию (смотрите объявления констант). Так что нам придётся позаботиться об изменении их размеров.

Так как на главной форме места уже не осталось, мы добавим к проекту *новую форму*, на которой и разместим все потребные для этого элементы управления (Рис. 3.28).

Рис. 3.28. Форма для изменения размеров сеток

Свойства формы:

```
BorderIcons=[biSystemMenu]
BorderStyle=bsDialog
Caption=' Введите размеры новых полей'
Name=frmNewFig
Label1: Caption=' Ширина поля:'
Label2: Caption=' Высота левого поля:'
Label3: Caption=' Числовые поля:'
Label4: Caption=' Высота верхнего поля:'
Label5: Caption=' Ширина левого поля:'
LblPoleWidth, lblPoleHeight: Caption='20'
LblTopHeight, lblLeftWidth: Caption='2'
```

Компоненты *TUpDown* TUpDown возьмите на странице *Win32 Палитры компонентов*:

```
UpDownPoleWidth, UpDownPoleHeigh:
```

Min=2
Orientation=udVertical
Position=2
UpDownTopHeight, UpDownLeftWidth:
Min=1

И кнопка *sbtOK* - для комплекта.

Процедуры, модуля *NewFigUnit* настолько просты, что любые комментарии к ним излишни:


```
unit NewFigUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ComCtrls, Buttons;

type
  TfrmNewFig = class(TForm)
    Label1: TLabel;
    Label2: TLabel;
    Label3: TLabel;
    Label4: TLabel;
    Label5: TLabel;
    lblPoleWidth: TLabel;
    lblPoleHeight: TLabel;
    lblTopHeight: TLabel;
    lblLeftWidth: TLabel;
    UpDownPoleWidth: TUpDown;
    sbtOK: TSpeedButton;
    UpDownPoleHeight: TUpDown;
    UpDownTopHeight: TUpDown;
    UpDownLeftWidth: TUpDown;
    procedure UpDownPoleWidthClick(Sender: TObject; Button: TUD-
BtnType);
    procedure sbtOKClick(Sender: TObject);
    procedure UpDownPoleHeightClick(Sender: TObject; Button: TUD-
BtnType);
    procedure UpDownTopHeightClick(Sender: TObject; Button: TUD-
BtnType);
    procedure UpDownLeftWidthClick(Sender: TObject; Button: TUD-
BtnType);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
```

```
  frmNewFig: TfrmNewFig;

implementation

uses NipponUnit;

{$R *.DFM}

//ИЗМЕНИТЬ ШИРИНУ ПОЛЯ
procedure TfrmNewFig.UpDownPoleWidthClick(Sender: TObject;
  Button: TUDBtnType);
begin
  lblPoleWidth.Caption:= inttostr(UpDownPoleWidth.Position);
end;
//ИЗМЕНИТЬ ВЫСОТУ ПОЛЯ
procedure TfrmNewFig.UpDownPoleHeightClick(Sender: TObject;
  Button: TUDBtnType);
begin
  lblPoleHeight.Caption:= inttostr(UpDownPoleHeight.Position);
end;
//ИЗМЕНИТЬ ВЫСОТУ ВЕРХНЕГО ЧИСЛОВОГО ПОЛЯ
procedure TfrmNewFig.UpDownTopHeightClick(Sender: TObject;
  Button: TUDBtnType);
begin
  lblTopHeight.Caption:= inttostr(UpDownTopHeight.Position);
end;
//ИЗМЕНИТЬ ШИРИНУ ЛЕВОГО ЧИСЛОВОГО ПОЛЯ
procedure TfrmNewFig.UpDownLeftWidthClick(Sender: TObject;
  Button: TUDBtnType);
begin
  lblLeftWidth.Caption:= inttostr(UpDownLeftWidth.Position);
end;
//УСТАНОВИТЬ НОВЫЕ РАЗМЕРЫ ПОЛЕЙ
procedure TfrmNewFig.sbtOKClick(Sender: TObject);
begin
  NipponUnit.POLE_WIDTH:= UpDownPoleWidth.Position;
  NipponUnit.POLE_HEIGHT:= UpDownPoleHeight.Position;
  NipponUnit.TOP_POLE_HEIGHT:= UpDownTopHeight.Position;
  NipponUnit.LEFT_POLE_WIDTH:= UpDownLeftWidth.Position;
  form1.Prepare(POLE_WIDTH,      POLE_HEIGHT,      TOP_POLE_HEIGHT,
LEFT_POLE_WIDTH);
  Close;
end;

end.
```


Вызываться эта форма вместе с «причиндалами» будет при нажатии на кнопку *sbtNewFig*:

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЕЙ
procedure TForm1.sbtNewFigClick(Sender: TObject);
begin
```

```
  frmNewFig.lblPoleWidth.Caption:= inttostr(POLE_WIDTH);
  frmNewFig.lblPoleHeight.Caption:= inttostr(POLE_HEIGHT);
  frmNewFig.lblLeftWidth.Caption:= inttostr(LEFT_POLE_WIDTH);
  frmNewFig.lblTopHeight.Caption:= inttostr(TOP_POLE_HEIGHT);
  frmNewFig.UpDownPoleWidth.Position:= POLE_WIDTH;
  frmNewFig.UpDownPoleWidth.Max:= MAX_POLE_WIDTH;
  frmNewFig.UpDownPoleHeight.Position:= POLE_HEIGHT;
  frmNewFig.UpDownPoleHeight.Max:= MAX_POLE_HEIGHT;
  frmNewFig.UpDownLeftWidth.Position:= LEFT_POLE_WIDTH;
  frmNewFig.UpDownLeftWidth.Max:= MAX_CELLNUM;
  frmNewFig.UpDownTopHeight.Position:= TOP_POLE_HEIGHT;
  frmNewFig.UpDownTopHeight.Max:= MAX_CELLNUM;
  frmNewFig.showmodal;
end;
```


Упомяните добрым словом в *NipponUnit* новый модуль:

```
implementation

uses NewFigUnit;
```

Впрочем, если вы забудете об этом, *Turbo Delphi* напомнит вам о ваших обязанностях.

Последние нововведения позволят вам запросто изменять размеры полей, как вашей душе будет угодно. Но нам предстоит ещё научиться вводить числа в числовые поля. Есть несколько способов для этого. Мы постараемся сделать этот процесс удобнее – для себя ведь делаем!

Каждому сверчку по шестку, или Нумеруем клетки

Входит и выходит –
замечательно выходит.

Каламбур Винни-Пуха

Отметим клетку, в которую будут вводиться числа, *красным* цветом. Красный квадратик можно перемещать точно так же, как и обычный курсор, но при переходе с одной клетки на другую нам придётся самим восстанавливать содержимое «покинутой» им клетки. Чтобы этот «указатель» не мешал нам при разгадывании задачи (ведь вводить числа нам будет незачем), проверяем, нажата ли кнопка *sbtDraw*. Если нажата, то указатель убираем.

```
//ПЕРЕМЕЩАТЬ ПОЗИЦИЮ ВВОДА В ВЕРХНЕМ ЧИСЛОВОМ ПОЛЕ
```

```
procedure TForm1.dgColsNumMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var ACol, ARow: Integer;
  r: TRECT;
  text: string;
begin
  //ACol, ARow <-- клетка с курсором
  dgColsNum.MouseToCell(x,y,ACol, ARow);
  //если курсор переместился на другую клетку -
  if (ACol<>cellMouseTop.x) or (ARow<>cellMouseTop.y) then
  begin
    //установить фокус ввода на верхнем поле:
    dgColsNum.SetFocus;
    //перерисовать предыдущую клетку с курсором:
    r:= dgColsNum.CellRect(cellMouseTop.x, cellMouseTop.y);
    dgColsNumDrawCell(self, cellMouseTop.x, cellMouseTop.y, R,
[]);
    //выделить красным цветом текущую клетку:
    if not sbtDraw.Down then begin
      r:= dgColsNum.CellRect(ACol, ARow);
      InflateRect(r,-1,-1);
      dgColsNum.Canvas.Brush.Color:= RGB(255,0,0);
      dgColsNum.Canvas.FillRect(R);
      //восстановить число в клетке:
      text:= masColsNum[ACol, ARow].sNum;
      with R, dgColsNum.Canvas do
       textrect(R, left+(right-left-textwidth(text)) div 2,
                top+(bottom-top-textheight(text)) div 2, text);
    end;
  end;
  //новая позиция ввода:
  cellMouseTop.x:= ACol;
  cellMouseTop.y:= ARow;
end; // dgColsNumMouseMove


//ПЕРЕМЕЩАТЬ ПОЗИЦИЮ ВВОДА В ЛЕВОМ ЧИСЛОВОМ ПОЛЕ
procedure TForm1.dgRowsNumMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var ACol, ARow: Integer;
  r: TRECT;
  text: string;
begin
  //ACol, ARow <-- клетка с курсором
  dgRowsNum.MouseToCell(x,y,ACol, ARow);
  if (ACol<>cellMouseLeft.x) or (ARow<>cellMouseLeft.y) then
  begin
    //установить фокус на левом поле:
    dgRowsNum.SetFocus;
    //перерисовать предыдущую клетку с курсором:
    r:= dgRowsNum.CellRect(cellMouseLeft.x, cellMouseLeft.y);
```

```
    dgRowsNumDrawCell(self, cellMouseLeft.x, cellMouseLeft.y, R,
[]);
    //выделить красным цветом текущую клетку:
    if not sbtDraw.Down then begin
      r:= dgRowsNum.CellRect(ACol, ARow);
      InflateRect(r,-1,-1);
      dgRowsNum.Canvas.Brush.Color:= RGB(255,0,0);
      dgRowsNum.Canvas.FillRect(R);
      text:= masRowsNum[ACol, ARow].sNum;
      with R, dgRowsNum.Canvas do
       textrect(R, left+(right-left-textwidth(text)) div 2,
                 top+(bottom-top-textheight(text)) div 2, text);
    end;
  end;
  cellMouseLeft.x:= ACol;
  cellMouseLeft.y:= ARow;
end; // dgRowsNumMouseMove
```


Запустите программу и погуляйте с курсором по *числовым* полям – красная клетка, как верный пёс, всюду будет следовать за вами (Рис. 3.29).

1	2	1	2	1	2	1	2	1	3	5	7	13	14	4	4	9	9	6	7	4	7	4	2	2
1	2	1	2	1	2	1	2	1			3	1	2	1	3	3	3	1	9	4	2	2	4	
2	5	7	9	4	2	1		5						5	5	2	6	8		3		1		
														3	2									
														4	2									

Рис. 3.29. Красная метка

Немного отдохнув, опять беритесь за дело. Для ввода чисел мы напишем процедуру, обрабатывающую событие *OnKeyPress*, возникающее при нажатии клавиши на клавиатуре. Прежде всего, проверим, не нажата ли кнопка *sbtDraw*, чтобы случайно не изменить числа при решении задачи. Затем нам нужно убедиться, что нажата именно *цифровая* клавиша, иначе возникнет ошибка при обработке введённых символов. Ну и последнее. Для подстраховки от неожиданных судорог и тиков мы ограничим длину вводимого числа *двумя* цифрами. Если всё в порядке, сохраняем число в массиве.

```
//ВВЕСТИ ЧИСЛО В ТЕКУЩУЮ КЛЕТКУ
procedure TForm1.dgColsNumKeyPress(Sender: TObject; var Key:
Char);
var
  s, text: string;
  rect: TRect;
begin
  if sbtDraw.Down then exit;
  if not(key in['0'..'9']) then exit;
  text:= Key;
```

```
  //если что-то уже есть в этой клетке -
  s:= masColsNum[cellMouseTop.x, cellMouseTop.y].sNum;
  //числа могут быть только однозначные и двузначные:
  if length(s)>=2 then exit;
  //записать число в клетку:
  masColsNum[cellMouseTop.x, cellMouseTop.y].sNum:= s+Key;
  rect:= dgColsNum.CellRect(cellMouseTop.x, cellMouseTop.y);
  dgColsNumDrawCell(self, cellMouseTop.x, cellMouseTop.y, Rect,
[]);
end; // dgColsNumKeyPress

//ВВЕСТИ ЧИСЛО В ЯЧЕЙКУ ЛЕВОГО ЧИСЛОВОГО ПОЛЯ
procedure TForm1.dgRowsNumKeyPress(Sender: TObject; var Key:
Char);
var
  s, text: string;
  rect: TRect;
begin
  if sbtDraw.Down then exit;
  if not(key in['0'..'9']) then exit;
  text:= Key;
  s:= masRowsNum[cellMouseLeft.x, cellMouseLeft.y].sNum;
  if length(s)>=2 then exit;
  masRowsNum[cellMouseLeft.x, cellMouseLeft.y].sNum:= s+Key;
  rect:= dgRowsNum.CellRect(cellMouseLeft.x, cellMouseLeft.y);
  dgRowsNumDrawCell(self, cellMouseLeft.x, cellMouseLeft.y, Rect,
[]);
end; // dgRowsNumKeyPress
```


А что делать, если вы запутались в цифрах? – Не мудрствуя лукаво, мы просто сотрём всё, что было в клетке, дважды щёлкнув на ней. Опять же для подстраховки проверяем, не решаем ли мы уже задачу, чтобы не усложнять себе и без того непростую жизнь.

```
//ОЧИСТИТЬ ТЕКУЩУЮ КЛЕТКУ
procedure TForm1.dgColsNumDblClick(Sender: TObject);
var Rect: TRECT;
begin
  if sbtDraw.Down then exit;
  masColsNum[cellMouseTop.x, cellMouseTop.y].sNum:='';
  rect:= dgColsNum.CellRect(cellMouseTop.x, cellMouseTop.y);
  dgColsNumDrawCell(self, cellMouseTop.x, cellMouseTop.y, Rect,
[]);
end;

//ОЧИСТИТЬ ТЕКУЩУЮ КЛЕТКУ
procedure TForm1.dgRowsNumDblClick(Sender: TObject);
var Rect: TRECT;
begin
  if sbtDraw.Down then exit;
  masRowsNum[cellMouseLeft.x, cellMouseLeft.y].sNum:='';
```

```
  rect:= dgRowsNum.CellRect(cellMouseLeft.x, cellMouseLeft.y);
  dgRowsNumDrawCell(self, cellMouseLeft.x, cellMouseLeft.y, Rect,
[]);
end;
```

Для проверки качества своей работы возьмём да и введём третью задачу из газеты *Поле чудес*, №3 за 1996 год. Это один из первых японских кроссвордов, появившихся у нас. Впрочем, вы можете взять и любой другой.

Запустите программу, нажмите самую первую кнопку (*sbtNewFig*) и в появившемся окне установите размеры всех трёх полей. Для нашего примера цифирь должна получиться такая (Рис. 3.30).

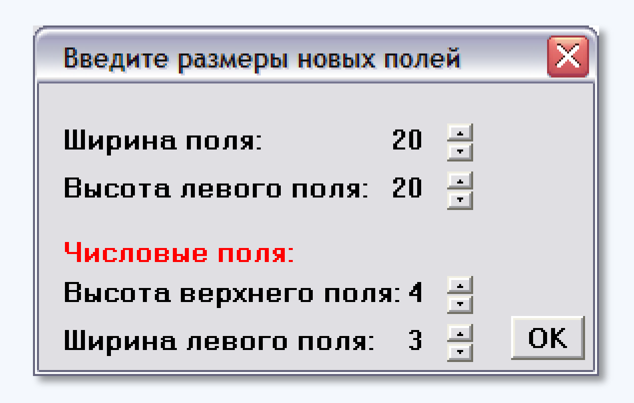


Рис. 3.30. Устнаваливаем размеры сеток

Нажав кнопку *ОК*, вы с удовольствием убедитесь, что игровое и числовые поля ужались и разжались как им велено. Теперь подводите красную метку к нужным клеткам (как сказал бы киношный Портос, *я вас умоляю*, не обижайте мышку, не кликайте ею всуе) и нажимайте на клавиатуре цифровые клавиши. Числа тут же появятся в клетках.

Как мы уже договорились, числа должны записываться в числовых полях от *верхнего* и *левого* краёв. А вот в газетах они, напротив, чаще записаны от *нижнего* и *правого*. Надо сказать, ввод чисел - удовольствие не из приятных, и вряд ли стоит усугублять его ещё и перетаскиванием чисел из одних клеток в другие. Поэтому мы сделаем для себя послабление и заставим программу гонять числа по полям, сами же будем спокойно вписывать числа в клетки, не придерживаясь твёрдых правил. Важно только сохранить последовательность чисел в рядах, а пустые клетки вы можете расставлять как угодно. Если у вас под рукой не оказалось первоисточника, то поверьте, что в итоге должна получиться такая «таблица» (Рис. 3.31).

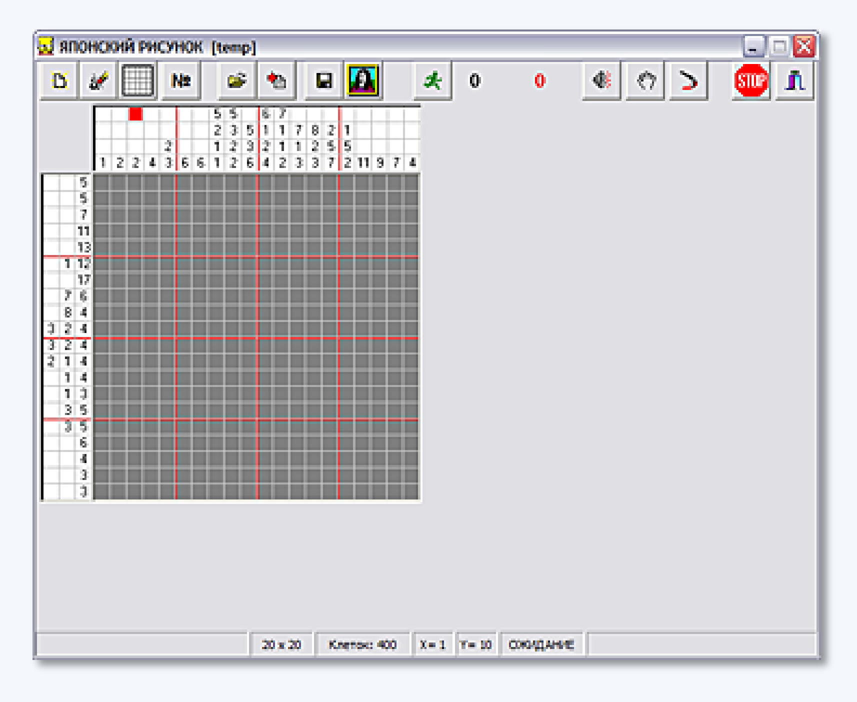


Рис. 3.31. Все числа – в клетках!

Этот способ ввода чисел, пожалуй, самый наглядный – легко сравнить задачу в газете с её «электронной копией», найти и исправить ошибки. Если же вам претит катать мышку по коврику, то вы можете набрать числа в обыкновенном текстовом редакторе, хотя бы в *Блокноте* или *WordPad'e*, которые входят в комплект операционной системы *Windows*.

Возможны *два* варианта создания текстовых файлов с условиями задачи. *Первый* - числа в строке должны отделяться друг от друга, по крайней мере, одним пробелом. *Второй* - числа пишутся слитно.

Рассмотрим сначала «пробельный» вариант. Формат файла такой. В *первой* строке - два числа: первое показывает *ширину* игрового поля, второе – его *высоту*. Во *второй* строке также два числа: первое задаёт *высоту верхнего* числового поля, второе – *ширину левого* числового поля. Затем записываются данные для *верхнего* числового поля. Каждая строка содержит числа из одной колонки. После чего аналогично записываются данные для *левого* числового поля, но в строках перечисляются числа из горизонтальных рядов. Например, информация о таком кроссворде (Рис. 3.32)

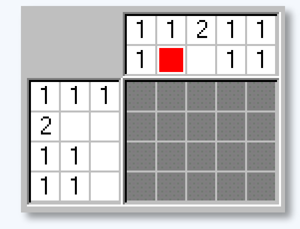


Рис. 3.32. «Образцовый» рисунок!

в текстовом исполнении будет выглядеть так:

```
5 4
2 3
1 1
1
2
1 1
1 1
1 1 1
2
1 1
1 1
```


Закончив ввод чисел, запишите файл как обычный текст без форматирования, а затем замените его расширение *TXT* другим - *JPN*. Конечно, можно оставить и *TXT*, но тогда в файл придётся вводить дополнительную информацию, чтобы отличить его при загрузке от обычного текста, иначе такие попытки будут приводить к ошибкам.

Чтобы не нажимать беспрестанно на клавишу *пробел*, вы можете воспользоваться другим способом ввода задач с клавиатуры. Формат файла в этом случае несколько иной (он используется в программе Гантверга и Самсонова *Soluter*, так что все претензии адресуйте им). *Первая* строка хранит *высоту* и *ширину* игрового поля. Информация о размерах числовых полей умалчивается. В следующих строках записаны числа – сначала из *левого* числового поля, затем из *верхнего*. Последовательность чисел в строках та же, что и раньше, но набираются они без пробелов:

```
45
111
2
11
11
11
1
2
11
11
```

Не забудьте изменить расширение файла на *JCP*.

Когда все числа в кроссворде меньше 10, этот формат, безусловно, удобнее для ввода задач. А вот б**о**льшие числа создают определённые трудности, так как число 10, например, при слитной записи будет трактоваться как два числа – 1 и 0. Так что придётся каждое число обозначать своим знач-

ком. В принципе, это делается просто: за цифрой *9* в таблице символов следует двоеточие - :, которое и будет заменять число 10. Следующие числа «шифруются» так:

```
11 - ;
12 - <
13 - =
14 - >
15 - ?
16 - @
17 – A
18 – B
19 – D
20 – E
```


И так далее. За всё надо платить - убрав пробелы, мы получили множество значков, в которых легко запутаться. Именно поэтому и был разработан первый формат текстового файла, который больше не используется ни в каких других программах, кроме нашей. Но вы можете придумать и что-нибудь своё – кто вам запретит!

Мотаем на ус, или Наш задачник

Наливай!

Из нашего

Вываливай!

Тоже

Теперь – тем или иным способом – вы сможете перенести задачу в компьютер. Встречаются же японские кроссворды почти в каждой головоломной газете. *Поле чудес*, *Магазин кроссвордов*, *Загадочная газета*, *Фаворит*, *Смекай!*, *Японский лабиринт* и многие другие редко обходятся без них. Имеются издания «с узкой специализацией», в которых ничего другого и нет. Например, газеты *Фудзияма*, *Японские кроссворды*, *Японский сканворд*. Но вы можете воспользоваться и готовыми задачами, которые прилагаются к уже упомянутым программам для самостоятельного решения. Причём большинство из них можно загрузить в нашу программу и решать непосредственно в ней. Правда, для этого нам сначала придётся создать необходимые процедуры для обработки «нестандартных» файлов. Так как это не относится непосредственно к нашей программе, то мы не будем подробно останавливаться на всех тонкостях чуждых нам форматов. При желании каждому из вас по силам самостоятельно в этом разобраться (а это

будет и небесполезно!). Пока же просто дважды щёлкните по кнопке *sbtLoadFig* и заполните заготовку процедуры следующим кодом:

```
//ЗАГРУЗИТЬ НОВУЮ ФИГУРУ
procedure TForm1.sbtLoadFigClick(Sender: TObject);
var
  s: string;
  F: TextFile;
  ss: array[1..MAX_POLE_WIDTH + MAX_POLE_HEIGHT] of string;
  nLines: integer;  //счётчик считанных строк фигуры
  maxLen: integer;
  i,j:integer;
  flgComm: boolean; //= TRUE, если считываем комментарий к фигуре
  w,h: integer;

  //ЗАГРУЗИТЬ ФАЙЛ JC {файл картинки программы Gun's Japanese Crossword)
  procedure LoadJcFile;
  var
    F: file of Byte;
    w,h: integer;
    b: byte;
    i,j:integer;
  begin
    {$i-}
    AssignFile(F, NameFig);
    Reset(F);
    {$i+}
    if IOResult<>0 then begin   {ошибка при загрузке файла}
      application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
      exit
    end;
    //вывести в заголовке формы имя загруженного файла:
    form1.caption:= NAME_PROG + '  [' + NameFig + ']';
    //начинаем считывать файл
    //считываем размеры поля
    //ширина поля - 4-ый байт:
    Seek(F, 4); Read(F, b); w:= b;
    //высота поля - 6-ой байт:
    Seek(F, 6); Read(F, b); h:= b;
    //проверить размеры фигуры:
    if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
      application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    POLE_WIDTH:= w; POLE_HEIGHT:= h;
    //проверяем длину файла:
    i:= FileSize(f);
    j:= 8 + POLE_HEIGHT * POLE_WIDTH;
    if i < j then begin
      application.MessageBox('Неверный размер файла!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    //загружаем данные для поля:
    for j:= 0 to POLE_HEIGHT-1 do
      for i:= 0 to POLE_WIDTH-1 do begin
        Seek(F, POLE_WIDTH*j+i + 8); Read(F, b);
         if b=0 then masPole[i,j]:= NUM_WHITE
```

```
      else masPole[i,j]:= NUM_BLACK;
    end;
  //закрыть файл - всё загружено:
  CloseFile(F);
  //оцифровать рисунок:
  sbtNumbersClick(Self);
  //очистить поле:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do masPole[i,j]:= NUM_GRAY;
  Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
end; //LoadJcFile

//ЗАГРУЗИТЬ ФАЙЛ JCW
procedure LoadJcwFile;
var
  F: file of Byte;
  i,j,n:integer;
  w,h: integer;
  b, pred: byte;
  bi, predi: integer;
  off: integer;
  ng: integer; //число групп в ряду
  p: TPole;
begin
  {$i-}
  AssignFile(F, NameFig);
  Reset(F);
  {$i+}
  if IOResult<>0 then begin   {ошибка при загрузке файла}
    application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
    exit
  end;
  //вывести в заголовке формы имя загруженного файла:
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  //начинаем считывать файл
  //считываем размеры поля
  //ширина поля - первый байт:
  Seek(F, 0); Read(F, b); w:= b;
  //высота поля - второй байт:
  Seek(F, 1); Read(F, b); h:= b;
  //проверить размеры фигуры:
  if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
    application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
    CloseFile(F); exit
  end;
  POLE_WIDTH:= w; POLE_HEIGHT:= h;
  //проверяем длину файла:
  i:= FileSize(f);
  j:= 2 + POLE_HEIGHT * POLE_WIDTH;
  if i < j then begin
    application.MessageBox('Неверный размер файла!',NAME_PROG, MB_OK);
    CloseFile(F); exit
  end;
  //считываем данные для верхнего числового поля:
  off:= 2;
  h:=0; //высота верхнего числового поля
  for i := 0 to POLE_WIDTH -1 do //- по длине поля
```

```
begin
  ng:= 0;   //число групп в ряду
  n:= 0;    //- длина группы
  pred:= 0; //- предыдущий байт
  //в каждом ряду должно быть хотя бы одно число:
  masColsNum[i,0].sNum:= '0';
  //для каждого столбца:
  for j := 0 to POLE_HEIGHT -1 do
  begin
    Seek(F, POLE_HEIGHT*i+j + off);
    //считать очередной байт:
    Read(F, b);
    //сохранить его:
    p[i,j]:= b;
    if b=2 then begin       //- чёрная клетка
      if pred<>2 then begin //- предыдущая клетка не чёрная -->
        //записать группу:
        masColsNum[i,ng-1].sNum:= inttostr(n);
        //начинается новая группа:
        inc(ng);
        n:= 0;
      end;
      //увеличить длину группы:
      inc(n);
    end;
    pred:= b;
  end;
  masColsNum[i,ng-1].sNum:= inttostr(n);
  masColsNum[i,-1].Num:= ng;
  if ng> h then h:= ng;
end;
//закрыть файл - всё загружено:
CloseFile(F);
//формируем данные для левого числового поля:
w:=0; //ширина левого числового поля
for j := 0 to POLE_HEIGHT -1 do //- по всем строкам поля
begin
  ng:= 0;    //число групп в ряду
  n:= 0;     //- длина группы
  predi:= 0; //- предыдущий байт
  //в каждом ряду должно быть хотя бы одно число:
  masRowsNum[0,j].sNum:= '0';
  for i := 0 to POLE_WIDTH -1 do //- по длине строки
  begin
    //считать очередное число:
    bi:= p[i,j];
    if bi=2 then begin //- чёрная клетка
      if predi<>2 then begin //- предыдущая клетка не чёрная -->
        //записать группу:
        masRowsNum[ng-1,j].sNum:= inttostr(n);
        //начинается новая группа:
        inc(ng);
        n:= 0;
      end;
      //увеличить длину группы:
      inc(n);
    end;
```

```
        predi:= bi;
      end; // i
      masRowsNum[ng-1,j].sNum:= inttostr(n);
      masRowsNum[-1,j].Num:= ng;
      if ng> w then w:= ng;
    end; //j
    //проверить число групп:
    if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
      application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    TOP_POLE_HEIGHT:= h;
    LEFT_POLE_WIDTH:= w;
    //вывести картинку на игровое поле:
    for j:= 0 to POLE_HEIGHT-1 do
      for i:= 0 to POLE_WIDTH-1 do
        case p[i,j] of
          2: masPole[i,j]:= NUM_BLACK;
          else masPole[i,j]:= NUM_WHITE;
        end;
    //вывести поля заданных размеров с числами:
    Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
  end;// LoadJcwFile

  //ЗАГРУЗИТЬ ФАЙЛ JPN
  procedure LoadJpnFile;
  var
    F: TextFile;
    i:integer;
    w,h: integer;
    ng: integer;    //число групп в ряду
    s: string;
    pos: integer;  //позиция в строке
    sNum: string;  //число
    //получить число из заданной строки
    function GetNumber(s: string; var pos: integer; var s2: string):
boolean;
    //= FALSE, если строка ещё не кончилась
    begin
      s2:= '';
      if pos> Length(s) then Result:= TRUE else Result:= FALSE;
      while (pos<= Length(s)) and (s[pos]<> ' ') do begin
        s2:= s2 + s[pos]; inc(pos)
      end;
      //пропустить пробелы:
      while (pos< Length(s)) and (s[pos]= ' ') do inc(pos);
      if (pos= Length(s)) and (s[pos]= ' ') then Result:= TRUE;
    end;
  begin
    {$i-}
    AssignFile(F, NameFig);
    Reset(F);
    {$i+}
    if IOResult<>0 then begin   {ошибка при загрузке файла}
      application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
      exit
    end;
```

```
//вывести в заголовке формы имя загруженного файла:
form1.caption:= NAME_PROG + '  [' + NameFig + ']';
//начинаем считывать файл
//считываем размеры поля
//ширина поля - первое число
//высота поля - второе:
Readln(F, S);
pos:= 1;
GetNumber(S, pos, sNum); w:= strtoint(sNum);
GetNumber(S, pos, sNum); h:= strtoint(sNum);
//проверить размеры фигуры:
if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
  application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
  CloseFile(F);
  exit
end;
POLE_WIDTH:= w; POLE_HEIGHT:= h;
//считываем размеры числовых полей
//высота верхнего числового поля первое число,
//ширина левого числового поля - второе:
Readln(F, S);
pos:= 1;
GetNumber(S, pos, sNum); h:= strtoint(sNum);
GetNumber(S, pos, sNum); w:= strtoint(sNum);
//проверить число групп:
if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
  application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
  CloseFile(F); exit
end;
TOP_POLE_HEIGHT:= h;
LEFT_POLE_WIDTH:= w;
//считываем данные для верхнего числового поля:
For i:= 0 to POLE_WIDTH-1 do begin
  if eof(f) then begin
    s:='Не хватает данных!';
    application.MessageBox(PChar(s),NAME_PROG,MB_OK );
    exit
  end;
  //считать строку из файла:
  Readln(F, S);
  //выделить числа из строки:
  pos:= 1; ng:= -1;
  while GetNumber(S, pos, sNum)= FALSE do begin
    inc(ng);
    masColsNum[i, ng].sNum:= sNum;
  end;
end;
//считываем данные для левого числового поля:
For i:= 0 to POLE_HEIGHT-1 do begin
  if eof(f) then begin
    s:='Не хватает данных!';
    application.MessageBox(PChar(s),NAME_PROG,MB_OK );
    exit
  end;
  //считать строку из файла:
  Readln(F, S);
  //выделить числа из строки:
```

```
      pos:= 1; ng:= -1;
      while GetNumber(S, pos, sNum)= FALSE do begin
        inc(ng);
        masRowsNum[ng,i].sNum:= sNum;
      end;
    end;
  //закрыть файл - всё загружено:
  CloseFile(F);
  //вывести поля заданных размеров с числами:
  Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
end;// LoadJpnFile

  //ЗАГРУЗИТЬ ФАЙЛ JPC из программы JpcWin Матвея Ильяшенко
  procedure LoadJpcFile;
  var
    F: file of Byte;
    i,j,n:integer;
    w,h: integer;
    b: byte;
    off: integer;
    function ReadWord(index: Longint): word;
    var
      b1, b2: byte;
    begin
      //считать первый байт:
      Seek(F, index); Read(F, b1);
      //считать второй байт:
      Seek(F, index+1); Read(F, b2);
      //вычислить слово:
      Result:= 256*b2+b1;
    end;
  begin
    //загрузить файл:
    {$i-}
    AssignFile(F, NameFig);
    Reset(F);
    {$i+}
    if IOResult<>0 then begin   {ошибка при загрузке файла}
      application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
      exit
    end;
    //вывести в заголовке формы имя загруженного файла:
    form1.caption:= NAME_PROG + '  [' + NameFig + ']';
    //начинаем считывать файл -
    //считываем размеры поля:
    w:= ReadWord(0);
    h:= ReadWord(2);
    //проверить размеры фигуры:
    if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
      application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    POLE_WIDTH:= w; POLE_HEIGHT:= h;
     //считываем размеры числовых полей:
    h:= ReadWord(4);
    w:= ReadWord(6);
    //проверить число групп:
```

```
  if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
    application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
    CloseFile(F); exit
  end;
  TOP_POLE_HEIGHT:= h;
  LEFT_POLE_WIDTH:= w;
  //проверяем длину файла:
  i:= FileSize(f);
  j:= 8+POLE_HEIGHT*LEFT_POLE_WIDTH + POLE_WIDTH* TOP_POLE_HEIGHT;
  if i<>j then begin
    application.MessageBox('Неверный размер файла!',NAME_PROG, MB_OK);
    CloseFile(F); exit
  end;
  //считываем данные для левого числового поля:
  off:= 8;
  for j := 0 to POLE_HEIGHT -1 do //сверху
  begin
    n:= 0;
    //в каждом ряду должно быть хотя бы одно число:
    masRowsNum[0,j].sNum:= '0';
    for i := LEFT_POLE_WIDTH - 1 downto 0 do //справа налево
    begin
      Seek(F, POLE_HEIGHT*i+j+ off);
      Read(F, b);
      if b<>0 then begin
        masRowsNum[n,j].sNum:= inttostr(b);
        inc(n);
      end;
    end;
  end;
  //считываем данные для верхнего числового поля:
  off:= LEFT_POLE_WIDTH * POLE_HEIGHT + 8;
  for i := 0 to POLE_WIDTH -1 do
  begin
    n:= 0;
    //в каждом ряду должно быть хотя бы одно число:
    masColsNum[i,0].sNum:= '0';
    for j := TOP_POLE_HEIGHT - 1 downto 0 do //снизу вверх
    begin
      Seek(F, POLE_WIDTH* j+ i+ off);
      Read(F, b);
      if b<>0 then begin
        masColsNum[i,n].sNum:= inttostr(b);
        inc(n);
      end;
    end;
  end;
  //закрыть файл - всё загружено:
  CloseFile(F);
  //вывести поля заданных размеров с числами:
  Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
end;// LoadJpcFile

//ЗАГРУЗИТЬ ФАЙЛ JCR из программы "Японский кроссворд 2000"
//Егоркина И.В.
procedure LoadJcrFile;
var
```

```
    F: file of Byte;
    i,j,n:integer;
    w,h: integer;
    b: byte;
    off: integer;
  begin
    //загрузить файл:
    {$i-}
    AssignFile(F, NameFig);
    Reset(F);
    {$i+}
    if IOResult<>0 then begin   //ошибка при загрузке файла
      application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
      exit
    end;
    //вывести в заголовке формы имя загруженного файла:
    form1.caption:= NAME_PROG + '  [' + NameFig + ']';
    //начинаем считывать файл -
    //считываем размеры поля:
    Seek(F, 0); Read(f,b); w:= b;
    Seek(F, 1); Read(f,b); h:= b;
    //проверить размеры фигуры:
    if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
      application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    POLE_WIDTH:= w; POLE_HEIGHT:= h;
    //считываем размеры числовых полей:
    Seek(F, 2); Read(f,b); w:= b;
    Seek(F, 3); Read(f,b); h:= b;
    //проверить число групп:
    if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
      application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    TOP_POLE_HEIGHT:= h;
    LEFT_POLE_WIDTH:= w;
    //проверяем длину файла:
    i:= FileSize(f);
    j:= 4+POLE_HEIGHT*LEFT_POLE_WIDTH + POLE_WIDTH* TOP_POLE_HEIGHT;
    if i<>j then begin
      application.MessageBox('Неверный размер файла!',NAME_PROG, MB_OK);
      CloseFile(F); exit
    end;
    //считываем данные для верхнего числового поля:
    off:= 4;
    for i := 0 to POLE_WIDTH -1 do
    begin
      n:= 0;
      //в каждом ряду должно быть хотя бы одно число:
      masColsNum[i,0].sNum:= '0';
      for j := TOP_POLE_HEIGHT - 1 downto 0 do //снизу вверх
      begin
        Seek(F, TOP_POLE_HEIGHT*i+j+ off);
        Read(F, b);
        if b<>0 then begin
          masColsNum[i,n].sNum:= inttostr(b);
```

```
          inc(n);
        end;
      end;
    end;
    //считываем данные для левого числового поля:
    off:= TOP_POLE_HEIGHT * POLE_WIDTH + 4;
    for j := 0 to POLE_HEIGHT -1 do //сверху вниз
    begin
      n:= 0;
      for i := LEFT_POLE_WIDTH - 1 downto 0 do
      begin
        Seek(F, LEFT_POLE_WIDTH *j+i+ off);
        Read(F, b);
        if b<>0 then begin
          masRowsNum[n,POLE_HEIGHT -1-j].sNum:= inttostr(b);
          inc(n);
        end;
      end;
    end;
    //закрыть файл - всё загружено:
    CloseFile(F);
    Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
  end;// LoadJcrFile

begin
  if status='ПОИСК' then exit;
  //"прокручиваем" все поля в начало:
  dgPole.LeftCol:= 0; dgPole.TopRow:= 0;
  dgColsNum.LeftCol:= 0; dgColsNum.TopRow:= 0;
  dgRowsNum.LeftCol:= 0; dgRowsNum.TopRow:= 0;
  //число разгаданных клеток = 0:
  lblReady.Caption:='0';
  //начнём решение задачи с нулевого уровня:
  lblLevel.Caption:='0';
  flgComm:= false;
  frmMemo.memo1.Clear;
  frmMemo.memo1.SetFocus;
  //файлы по умолчанию имеют расширение jcp:
  opendialog1.DefaultExt:='jcp';
  opendialog1.Filter:=
    'Japan puzzle(*.jcp, *.jpc, *.jcw, *.jpn, *.jc, *.jcr)|'+
    '*.JCP;*.JPC;*.JCW; *.JPN; *.JC; *.JCR';
  //ищем файлы в каталоге 'FIGURE':
  s:=extractfilepath(application.exename)+'FIGURE\';
  opendialog1.InitialDir:= s;
  opendialog1.Title:='Загрузите новую фигуру';
  if opendialog1.Execute then begin
    //очистить игровое и числовые поля:
    sbtClearPoleClick(self);
    //выбрали файл с именем NameFig=s:
    s:= opendialog1.filename;
    NameFig:=s;
    //файл формата JPC
    s:= ExtractFileExt(NameFig);
    if s= '.jpc' then begin LoadJpcFile; exit end
      //файл формата JCW - задания для самостоятельного решения
      //из программы Романа Гантверга и Дмитрия Самсонова Japan Crossword
```

```
      //Soluter (Японский кроссворд - Решатель), 1999
    else if s= '.jcw' then begin LoadJcwFile; exit end
      //обычный текстовый файл:
    else if s= '.jpn' then begin LoadJpnFile; exit end
      //файл картинки программы Gun's Japanese Crossword:
    else if s= '.jc' then begin LoadJcFile; exit end
    //файл задачи из программы "Японский кроссворд 2000":
    else if s= '.jcr' then begin LoadJcrFile; exit end;
    {$i-}
    AssignFile(F,NameFig);
    Reset(F);
    {$i+}
    if IOResult<>0 then begin  //ошибка при загрузке файла
      application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
      exit
      end;
    //начинаем считывать файл:
    nLines:=0; //считано строк фигуры
    while not eof(f) do begin
      //считать строку из файла:
      Readln(F, S);
      //комментарий к фигуре?
      if ((length(s)>1) and (s[1]='/') and (s[2]='/')) or
        (flgComm= true) then    //- комментарий
      begin
        if flgComm=false then   //- начало комментария
        begin
          flgComm:= true;
          s:=copy(s,3,length(s));
          frmMemo.memo1.Lines.Text:=s;
        end
        else
          frmMemo.memo1.Lines.Text:=frmMemo.memo1.Lines.Text+#10+s;
      end
      else begin //- строка фигуры
        inc(nLines);
        //сохранить строку в массиве:
        ss[nLines]:=s;
      end
    end;
    //закрыть файл - всё загружено:
    CloseFile(F);
    //размеры игрового поля -
    //высота поля в клетках:
    h:= ord(ss[1][1])-ord('0');
    //ширина поля в клетках:
    w:= ord(ss[1][2])-ord('0');
    //проверить размеры фигуры:
    if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
      application.MessageBox('Слишком большая фигура!',NAME_PROG, MB_OK);
      exit
    end;
    //количество строк для числовых полей должно соответствовать
    //размерам поля:
    if nLines-1 <> h + w then begin
      application.MessageBox('Неверное   количество   данных!',NAME_PROG,
MB_OK);
```

```
      exit
    end;
    //занести данные в masColsNum, masRowsNum:
    maxLen:=0;
    for j:=2 to h+1 do begin
      s:=ss[j];  //- очередная строка из массива
      //по длине очередной строки -
      for i:=1 to length(s) do begin
        if length(s)> maxLen then maxLen:= length(s);
        //проверить количество групп чисел в строке:
        if maxLen > MAX_CELLNUM then begin
          application.MessageBox('Слишком    много    групп!',NAME_PROG,
MB_OK);
          //очистить masRowsNum[]:
          Clear_masRowsNum;
          exit
        end;
        //занести число в массив:
        masRowsNum[i-1,j-2].sNum:= inttostr(ord(s[i])-ord('0'));
      end;
    end;
    //макс. число групп в левом числовом поле:
    LEFT_POLE_WIDTH:= maxLen;
    //начинаем заполнение массива верхнего числового поля:
    maxLen:=0;
    for j:=h+2 to nLines do begin
      //очередная строка:
      s:=ss[j];
      //по длине очередной строки -
      for i:=1 to length(s) do begin
        if length(s)> maxLen then maxLen:= length(s);
        //проверить количество групп чисел в строке:
        if maxLen > MAX_CELLNUM then begin
          application.MessageBox('Слишком    много    групп!',NAME_PROG,
MB_OK);
          //очистить masColsNum[]:
          Clear_masColsNum;
          exit
        end;
        //занести число в массив:
        masColsNum[j-h-2,i-1].sNum:= inttostr(ord(s[i])-ord('0'));
      end;
    end;
    //макс. число групп в верхнем числовом поле:
    TOP_POLE_HEIGHT:= maxLen;
    //размеры поля:
    POLE_WIDTH:= w; POLE_HEIGHT:= h;
    //вывести в заголовке формы имя загруженного файла:
    form1.caption:= NAME_PROG + '  [' + NameFig + ']';
    //вывести поля заданных размеров с числами:
    Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
  end; //opendialog1.Execute
end;//sbtLoadFigClick}
```

Следует заметить, что, как ни старайся последовательно излагать «материал», где-нибудь да и встретится анахронизм. Вот и в этой процедуре

встречаются «воспоминания о будущем», поэтому давайте забежим чуточку вперёд и, нарушив логику событий, сделаем два добрых дела.

Во-первых, добавим к проекту ещё одну форму – *frmMemo*, и сверху слева поместим компонент *Image1*, в котором будет отображаться растровая картинка или значок – они послужат нам при составлении собственных задач. Как и прежде, в компоненте *Memo1* вы можете записывать свои комментарии к картинкам (например, сложность решения, авторство задачи, источник и тому подобное) - если будет желание. Они сохранятся с условием задания и будут вместе с ним загружаться. Расположение компонентов на форме большого значения не имеет, положитесь на свой вкус. Можно хотя бы так (Рис. 3.33).

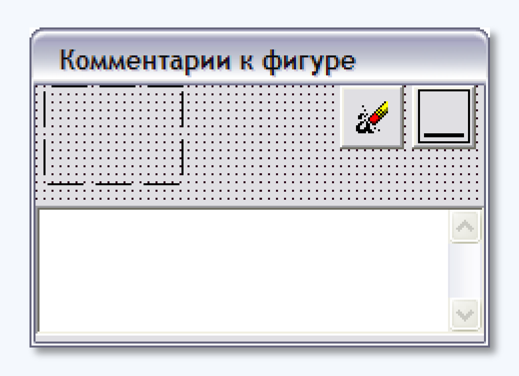


Рис. 3.33. Форма для комментариев

В тексте модуля *MemoUnit* нет ничего неожиданного для вас:

```
unit MemoUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, Buttons, ExtCtrls;

type
  TfrmMemo = class(TForm)
    Memo1: TMemo;
    sbtClear: TSpeedButton;
    sbtMinimize: TSpeedButton;
    Image1: TImage;
    procedure sbtClearClick(Sender: TObject);
    procedure sbtMinimizeClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
  end;

var
  frmMemo: TfrmMemo;

implementation

uses NipponUnit;

{$R *.DFM}

procedure TfrmMemo.sbtClearClick(Sender: TObject);
begin
  frmMemo.memo1.Clear;
  frmMemo.memo1.SetFocus
end;

procedure TfrmMemo.sbtMinimizeClick(Sender: TObject);
begin
  frmMemo.WindowState:=wsMinimized
end;

end.
```


Во-вторых, обеспечим работой кнопку *sbtNumbers*. Она потребуется нам позднее, когда мы займёмся разработкой собственных задач. Нарисовав картинку, вы просто нажмёте эту кнопку, и программа автоматически установит нужные размеры числовых полей и заполнит их числами. Так что вам никогда не придётся считать чёрные и белые клетки на своём рисунке: ваше дело – творчество, а считать должен компьютер! Действие этой процедуры достаточно прозрачно, что отнюдь не мешает вам «покопаться» в ней.

```
//ОЦИФРОВАТЬ РИСУНОК
procedure TForm1.sbtNumbersClick(Sender: TObject);
var
  i,j: integer;
  h,w: integer;
  n,ng: integer;
  Pred, len: integer;
begin
  //очистить цифровые поля:
  Clear_masColsNum;
  Clear_masRowsNum;
  //считываем данные для верхнего числового поля:
  h:=0; //- высота верхнего числового поля
  for i := 0 to POLE_WIDTH -1 do //- по длине поля
  begin
    ng:= 0;  //число групп в ряду
    len:= 0; //- длина группы
    Pred:= NUM_WHITE; //- предыдущая клетка
    //в каждом ряду должно быть хотя бы одно число:
```

```
    masColsNum[i,0].sNum:= '0';
    //для каждого столбца:
    for j := 0 to POLE_HEIGHT -1 do
    begin
      //считать очередной число:
      n:= masPole[i,j];
      if n= NUM_BLACK then begin //- чёрная клетка
          if pred<>NUM_BLACK then begin //- предыдущая клетка не чёрная -
->
            //записать группу:
            masColsNum[i,ng-1].sNum:= inttostr(len);
            //начинается новая группа:
            inc(ng);
            len:= 0;
          end;
          //увеличить длину группы:
          inc(len);
        end;
        pred:= n;
      end;
      masColsNum[i,ng-1].sNum:= inttostr(len);
      masColsNum[i,-1].Num:= ng;
      if ng> h then h:= ng;
    end;

    //формируем данные для левого числового поля:
    w:=0; //ширина левого числового поля
    for j := 0 to POLE_HEIGHT -1 do //- по всем строкам поля
    begin
      ng:= 0;   //- число групп в ряду
      len:= 0;  //- длина группы
      pred:= 0; //- предыдущее число
      //в каждом ряду должно быть хотя бы одно число:
      masRowsNum[0,j].sNum:= '0';
      for i := 0 to POLE_WIDTH -1 do //- по длине строки
      begin
        //считать очередное число:
        n:= masPole[i,j];
        if n= NUM_BLACK then begin //- чёрная клетка
          if pred<> NUM_BLACK then begin //-предыдущая клетка не чёрная -
->
            //записать группу:
            masRowsNum[ng-1,j].sNum:= inttostr(len);
            //начинается новая группа:
            inc(ng);
            len:= 0;
          end;
          //увеличить длину группы:
          inc(len);
        end;
        pred:= n;
      end; // i
      masRowsNum[ng-1,j].sNum:= inttostr(len);
      masRowsNum[-1,j].Num:= ng;
      if ng> w then w:= ng;
    end; //j
    //проверить число групп:
```

```
    if (w > MAX_CELLNUM) or (h > MAX_CELLNUM) then begin
      application.MessageBox('Слишком много групп!',NAME_PROG, MB_OK);
      exit
    end;
    TOP_POLE_HEIGHT:= h;
    LEFT_POLE_WIDTH:= w;
    //вывести поля заданных размеров с числами:
    Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT, LEFT_POLE_WIDTH);
end; // TForm1.sbtNumbersClick
```

Если вы ввели данные для числовых полей, то было жалко потерять их после завершения работы с программой. Но нас уже давно дожидается временно безработная (но с определённым местом жительства!) кнопка *sbtSaveFig*. Она-то и будет переписывать условие задачи в файл.

Нажав эту кнопку, вы попадёте в диалог, где сможете выбрать *название* файла для своей задачи. Конечно, лучше со смыслом, чем с порядковым номером, иначе потом её трудно будет отыскать среди множества других. Вместе с данными в файл будет помещён и комментарий, поэтому у вас есть возможность добавить свои замечания к задаче, которые потом могут вам пригодиться.

Мы не будем изобретать собственного формата файла, а воспользуемся тем, что и в программе *Soluter* Гантверга и Самсонова - *JCP*. Это позволит вам решать свои задачи не только в рассматриваемой программе, но и в некоторых других. Польза от этого такая. Если вы будете совершенствовать *нашу* программу, то вам легко будет сравнить её «убойную силу» не только с ее предыдущими версиями, но и с чужими достижениями.

```
//ЗАПИСАТЬ ЗАДАЧУ
procedure TForm1.sbtSaveFigClick(Sender: TObject);
var
  F: textfile;
  fn,s: string;
  i,j: integer;
begin
  if status='ПОИСК' then exit;
  //расширение файлов фигур:
  savedialog1.DefaultExt:='jcp';
  savedialog1.Filter:='Japan puzzle (*.jcp)|*.JCP';
  //записываем в каталог 'FIGURE':
  s:=extractfilepath(application.exename)+'FIGURE\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите фигуру на диск';
  savedialog1.filename:= NameFig;
  if not savedialog1.Execute then exit;
  //имя конечного файла:
  fn:= savedialog1.filename;
  //изменить расширение файла, если при записи было выбрано другое имя:
  fn:=ChangeFileExt(fn, '.jcp');
  NameFig:=fn;
```

```
  assignfile(f,fn);
  rewrite(f);
  //записать фигуру -
  //высота и ширина фигуры:
  writeln (f, chr(POLE_HEIGHT + ord('0')) + chr(POLE_WIDTH+ ord('0')));
  //данные из левого числового поля-
  //очередная строка:
  for j:= 0 to POLE_HEIGHT-1 do begin
    for i:= 0 to LEFT_POLE_WIDTH-1 do
      if masRowsNum[i,j].sNum<>'' then
        write (f, chr(strtoint(masRowsNum[i,j].sNum) + ord('0')));
    writeln(f, '');
  end;
  //данные из верхнего числового поля -
  //очередная строка:
  for j:=0 to POLE_WIDTH-1 do begin
    for i:= 0 to TOP_POLE_HEIGHT-1 do
      if masColsNum[j,i].sNum<>'' then
        write (f, chr(strtoint(masColsNum[j,i].sNum) + ord('0')));
    writeln(f, '');
  end;

  //записать комментарий к фигуре:
  s:=frmMemo.memo1.Lines.Text;
  if s<>'' then begin
    s:='//'+s;
    writeln (f,s)
  end;
  closefile(f);
  //вывести в заголовке формы новое название файла:
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  form1.Refresh ;
end; //Save
```


Делайте с нами, или Тонкие штучки на компьютере

Помнят руки!

Радостный возглас героя Юрия Никулина
из фильма *Когда деревья были большими*

Подготовив плацдарм, нам ничего другого не остаётся, как испытать свои силы в ручном решении – хотя и на компьютере – одного из японских кроссвордов. Вы можете начать с того самого рисунка, который был использован нами как пример решения задач. Тогда вы всегда сможете проконтролировать свои шаги и лучше усвоить *Правила*.

Само решение мы сделаем точно таким же, как и на бумаге, чтобы не морочить себе голову компьютерной премудростью.

1. Отыскав подходящий ряд, мы должны отметить разгаданные белые и чёрные клетки (напомню, что не разгаданные клетки - *серые*). Естественно использовать для этого левую и правую кнопки мыши. *Левой* мы будем ставить *чёрные* клетки, *правой* – *белые*. А вот если потребуется вернуть клетке её первоначальный *серый* цвет, то надо бы нажать *среднюю* кнопку мыши, да не все мышки так далеко продвинуты, поэтому серые клетки мы будем ставить той же *левой* кнопкой, но с нажатой клавишей **Shift**. Делать это придётся нечасто (не ошибайтесь при решении – и вовсе не придётся), так что смиритесь. Для пущего форсу вид мышиного курсора будет изменяться: при передвижении картинки он приобретёт вид загребущей *руки*, при рисовании – *кисти*. Причём передвигать картинку или рисовать на поле можно, только если нажата соответствующая кнопка в меню. Процесс рисования выглядит так: по координатам мышки в пикселях мы определяем, в какой клетке поля она находится, и заносим в массив код нужного цвета, после чего перерисовываем клетку в обновлённом виде.


```
//НАЖАТЬ КНОПКУ МЫШКИ
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  area: tRect;
begin
  if status='ПОИСК' then exit;
  //изменить форму курсора
  //передвигаем картинку:
  if sbtMove.down then begin
    screen.Cursor:=TCursor(crMove);
    dgPole.Cursor :=TCursor(crMove);
    screen.Cursor:=TCursor(crDefault);
    exit end;
  //рисуем на поле:
  if not sbtDraw.Down then exit;
  screen.Cursor:=TCursor(crKistj);
  dgPole.Cursor :=TCursor(crKistj);
  screen.Cursor:=TCursor(crDefault);

  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  cellmouse.x:=ACol;
  cellmouse.y:=ARow;
  area:= dgPole.CellRect(ACol, ARow);
  //если кнопка мыши нажата вместе с клавишей Shift,
  //то поставить серую клетку:
  if ssShift in shift then begin
    masPole[ACol, ARow]:=NUM_GRAY;
    //закрасить клетку:
    dgPoleDrawCell(self, ACol, ARow, area, []);
    exit
  end;
  //если нажата только левая кнопка - чёрная клетка:
  if ssLeft in shift then
  begin
    //занести в массив цвет клетки
    masPole[ACol, ARow]:=NUM_BLACK;
    //закрасить клетку:
```

```
    dgPoleDrawCell(self, ACol, ARow, area, []);
 end;
  //если нажата правая кнопка - белая клетка:
  if ssRight in shift then
  begin
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:= NUM_WHITE;
    //закрасить клетку:
    dgPoleDrawCell(self, ACol, ARow, area, []);
  end;
end; // TForm1.dgPoleMouseDown
```


При отпускании кнопки мыши мы просто возвращаем курсору обычный вид:

```
//ОТПУСТИТЬ КНОПКУ МЫШИ
procedure TForm1.dgPoleMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  //изменить форму курсора:
  screen.Cursor:=TCursor(crHand);
  dgPole.Cursor :=TCursor(crHand);
  screen.Cursor:=TCursor(crDefault);
end;
```

В принципе, этого вполне достаточно для того, чтобы решать задачи. Вот только не очень удобно щёлкать на каждой клетке, если разгадана целая группа клеток, а то и весь ряд целиком. Поэтому наделим мышку, как и курьёзного Микки Мауса, способностью закрашивать клетки при передвижении по полю. События здесь происходят точно те же, что и при щелчке. Мы добавим в процедуру *dgPoleMouseMove* ещё и возможность *циклически сдвигать* рисунок. Пока нам это ни к чему, но, взявшись за рисование собственных кроссвордов, вы оцените её сполна: если к уже готовой физиономии вам захочется пририсовать усы, а места для них нет, то просто сдвиньте картинку и потешьте себя. Поскольку рассуждать о сдвигах - хотя бы и поля – сейчас не очень уместно, то разберитесь в этом вопросе самостоятельно, тем более что здесь всё просто (такие «миражи» и «вертижи» самомнения часто возникают, когда программа уже отлажена).

```
//ПЕРЕМЕЩАТЬ МЫШКУ ПО ПОЛЮ
procedure TForm1.dgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var
  ACol,ARow: integer;
  area: TRect;

  //прокрутить поле влево
  procedure PoleLeft;
```

```
var
  n, x, y: Integer;
begin
  //сдвигаем:
  For y := 0 To POLE_HEIGHT-1 do begin
    //запомнить цвет первой клетки:
    n:= masPole[0,y];
    For x := 0 To POLE_WIDTH - 2 do
      masPole[x,y] := masPole[x+1,y];
    //последний ряд <-- первый ряд:
    masPole[POLE_WIDTH-1,y]:= n;
  end
end; //PoleLeft
//Сдвинуть поле вправо
procedure PoleRight;
var
  n, x,y: Integer;
  XR: Integer; //правый столбец
begin
  XR:= POLE_WIDTH-1;
  For y := 0 To POLE_HEIGHT-1 do begin
    n:= masPole[XR, y];
    For x := XR downTo 1 do
      masPole[x,y]:= masPole[x-1,y];
    //первый ряд <-- последний ряд
    masPole[0, y]:=n;
  end
End; //PoleRight
//Сдвинуть поле вверх
procedure PoleUp;
var
  n, x, y: Integer;
begin
  For x := 0 To POLE_WIDTH-1 do begin
    n:= masPole[x,0];
    For y := 1 To POLE_HEIGHT-1 do
      masPole[x,y-1] := masPole[x,y];
    //нижняя строка <-- верхняя строка:
    masPole[x, POLE_HEIGHT-1]:= n;
  end
End; //PoleUp
//Сдвинуть поле вниз
procedure PoleDown;
var
  n, x, y: Integer;
begin
  For x := 0 To POLE_WIDTH-1 do begin
    n:= masPole[x, POLE_WIDTH-1];
    For y := POLE_HEIGHT-1 downTo 1  do
      masPole[x,y] := masPole[x,y-1];
    //верхняя строка <-- нижняя строка:
    masPole[x, 0]:= n;
  end
```

```
  End; //PoleDown

begin
  if status='ПОИСК' then exit;
  dgPole.SetFocus;
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  statusbar1.Panels[3].text:='X= '+inttostr(ACol+1);
  statusbar1.Panels[4].text:='Y= '+inttostr(ARow+1);
  //циклически сдвигаем картинку:
  if (sbtMove.Down) and (ssLeft in shift)  then begin
    if cellmouse.x>ACol then PoleLeft;
    if cellmouse.x<ACol then PoleRight;
    if cellmouse.y>ARow then PoleUp;
    if cellmouse.y< ARow then PoleDown;
    dgPole.Invalidate;
    cellmouse.x:=ACol;
    cellmouse.y:=ARow;
  end;
  //рисуем:
  if not sbtDraw.Down then exit;
  area:= dgPole.CellRect(ACol, ARow);
  //если кнопка мыши нажата вместе с клавишей Shift,
  //то поставить серую клетку:
  if (ssShift in shift) and (ssLeft in shift) and
     ((cellmouse.x<>ACol) or (cellmouse.y<>ARow)) then
  begin
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:=NUM_GRAY;
    //закрасить клетку:
    dgPoleDrawCell(Self, ACol, ARow, area, []);
    cellmouse.x:=ACol;
    cellmouse.y:=ARow;
    exit
  end;
  //если нажата левая кнопка - чёрная клетка
  if (ssLeft in shift) and
     ((cellmouse.x<>ACol) or (cellmouse.y<>ARow))then
  begin
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:=NUM_BLACK;
    //закрасить клетку:
    dgPoleDrawCell(Self, ACol, ARow, area, []);
  end;
  if (ssRight in shift) and
     ((cellmouse.x<>ACol) or (cellmouse.y<>ARow))then
  begin
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:= NUM_WHITE;
    //закрасить клетку:
    dgPoleDrawCell(Self, ACol, ARow, area, []);
  end;
  //новые координаты мыши:
```

```
  cellmouse.x:=ACol;
  cellmouse.y:=ARow;
end; // TForm1.dgPoleMouseMove
```

2. Помните, как в рассмотренном в начале главы примере мы отмечали разгаданные группы *кружком*? Казалось бы, мелочь, а сильно облегчает жизнь: сразу видно, какие группы ещё не разгаданы. Сейчас мы поступим иначе – кружки ставить не будем, а вместо этого пометим разгаданные группы *жёлтым* цветом. Иначе говоря, мы изменяем при этом *статус* группы – из состояния «неразгаданности» *stWhite*, мы переводим её в более приятное состояние *stYellow* (щёлкнув еще раз, вы вернёте клетке белый цвет). Так что процедуре, обрабатывающей нажатие кнопки в числовом поле, нужно просто занести нужное значение статуса в «кликнутую» клетку:


```
//ИЗМЕНИТЬ СТАТУС ГРУППЫ
procedure TForm1.dgColsNumClick(Sender: TObject);
var ACol, ARow: Integer;
  r: TRECT;
begin
  if not sbtDraw.Down then exit;
  ACol:= cellMouseTop.x; ARow:= cellMouseTop.y;
  case masColsNum[ACol, ARow].StatusGroup of
    stYellow: masColsNum[ACol, ARow].StatusGroup:= stWhite;
    stWhite: masColsNum[ACol, ARow].StatusGroup:= stYellow;
  end;
  r:= dgColsNum.CellRect(ACol, ARow);
  dgColsNumDrawCell(self, ACol, ARow, r, []);
end;

//ИЗМЕНИТЬ СТАТУС ГРУППЫ
procedure TForm1.dgRowsNumClick(Sender: TObject);
var ACol, ARow: Integer;
  r: TRECT;
begin
  if not sbtDraw.Down then exit;
  ACol:= cellMouseLeft.x; ARow:= cellMouseLeft.y;
  case masRowsNum[ACol, ARow].StatusGroup of
    stYellow: masRowsNum[ACol, ARow].StatusGroup:= stWhite;
    stWhite: masRowsNum[ACol, ARow].StatusGroup:= stYellow;
  end;
  r:= dgRowsNum.CellRect(ACol, ARow);
  dgRowsNumDrawCell(self, ACol, ARow, R, []);
end;
```

В примере мы дополнительно отмечали и *разгаданные* ряды, здесь мы этого делать не будем – и так всё хорошо видно. А кому не видно – измените процедуру так, чтобы при клике на жёлтой клетке она перекрашивалась в *зелёную* (переход в состояние *stGreen*).

Теперь решение задачи на компьютере мало чем отличается от обычного, поэтому можете любую задачу решать и так, и этак (Рис. 3.34).

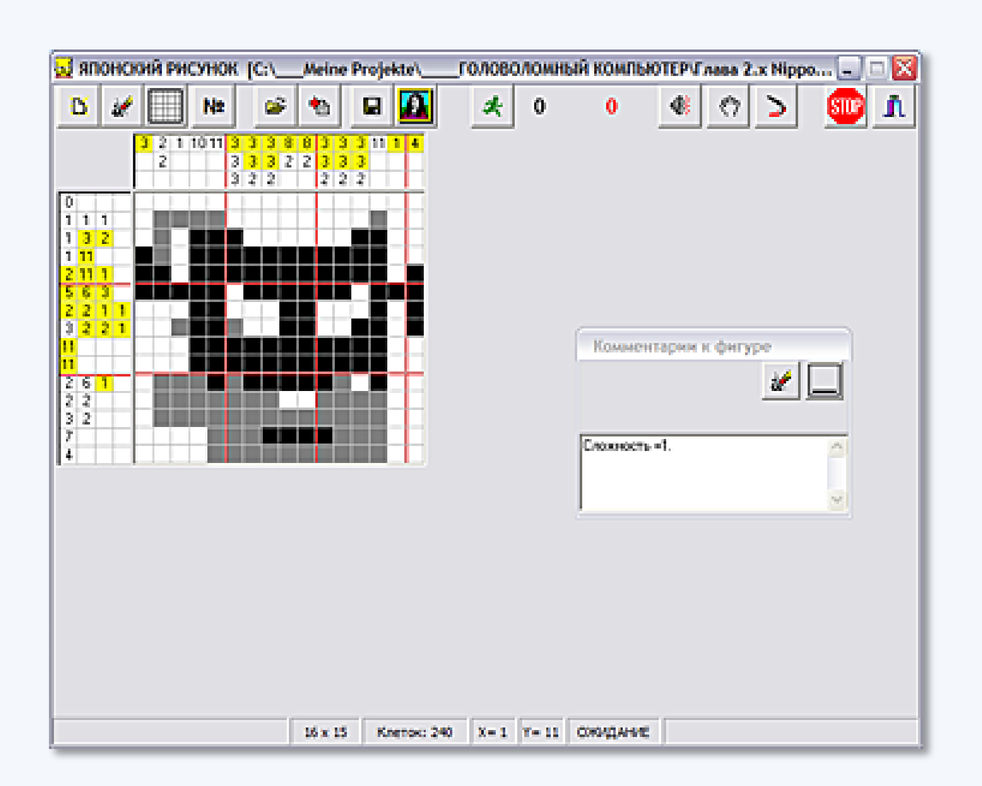


Рис. 3.34. Осторожно: задача решается автоматически!

На этом мы и остановимся: решать задачи вручную – это *не наш метод*, как сказал однажды верзила Федя вечному студенту Шурику. Если же эта мысль не оставляет вас в покое, то вам нужно позаботиться о запоминании позиции на поле после каждого хода (как это сделать, будет рассказано дальше) и о возврате назад, если вы попадёте в тупик. Клетку, в цвете которой вы не уверены, следует помечать иначе, чем разгаданные и не разгаданные, чтобы потом можно было бы вернуться к ней. Неплохо было бы подсчитывать количество уже разгаданных клеток. Не очень удобно решать на экране задачи больших размеров, так как ряды целиком не видны. Придётся постоянно двигать поле. Можно немного улучшить ситуацию, уменьшив размеры клеток в сетках и, соответственно, размер шрифта. Но тут уж берегите глаза – решение огромных задач потребует от вас нескольких часов работы и напряжённого вглядывания в экран. Так что если вас неудержимо тянет к монументальным формам, купите себе монитор подходящих размеров. Конечно, для воплощения всего этого придётся потрудиться, но – охота пуще неволи!

А нам пора дальше. Заставим компьютер делать всё, чему мы уже научились сами. А учить компьютер – вот где настоящая педагогическая поэма! Испытайте себя в роли учителя – и вы много нового откроете для себя (и в себе).

Компьютерная педагогика, или Пускай работает Иван

Не можешь – научим,
Не хочешь – заставим!

Из книги *Армейская жизнь как таковая*

Какое наслаждение наблюдать за тем,
как работают другие!

Из *Манифеста* Филона

Прежде чем с головой окунуться в пучину страсти, мы сначала должны проверить условие задачи, иначе всё решение пойдёт насмарку. Для этого мы напишем функцию *Testing*, которая будет возвращать значение *True*, если всё правильно, или *False*, если обнаружились ошибки.

Во-первых, она должна проверить, а загружена ли задача. Это легко установить по названию задачи, которое хранится в строковой переменной *NameFig*. Если её значение равно *'temp'*, то это ошибка: задача не загружена. Эта же ситуация возникнет, если вы ввели данные задачи в числовые поля, но не сохранили её на диске под другим именем.

Во-вторых, если вы *вводили* числа в поля, то нужно переписать их так, чтобы они были записаны правильно, то есть от левого и верхнего краёв числовых полей. За комфорт при вводе чисел придётся расплачиваться написанием процедуры *MoveNums*. В ней все числа будут сдвинуты вверх и влево (естественно, если числа сразу были записаны верно, то ничего и не изменится). И не забудьте объявить процедуру в разделе *private* типа формы:

```
procedure MoveNums;
```

```
//ПЕРЕНЕСТИ ЧИСЛА В ЧИСЛОВЫХ ПОЛЯХ
procedure TForm1.MoveNums;
var
  i, j, n: integer;
  s: string;
begin
  //переносим все числа к верхней границе сетки
  for i:= 0 to POLE_WIDTH-1 do
  begin
    n:= 0;
    //каждый столбец:
    for j:= 0 to TOP_POLE_HEIGHT-1 do
    begin
      s:= masColsNum[i,j].sNum;
```

```
      if s <> '' then begin
        masColsNum[i,j].sNum:= '';
        masColsNum[i,n].sNum:= s;
        inc(n);
      end;
    end;
  end;
  //левое числовое поле:
  for i:= 0 to POLE_HEIGHT-1 do
  begin
    n:= 0;
    //в каждой строке:
    for j:= 0 to LEFT_POLE_WIDTH-1 do
    begin
      s:= masRowsNum[j,i].sNum;
      if s <> '' then begin
        masRowsNum[j,i].sNum:= '';
        masRowsNum[n,i].sNum := s;
        inc(n);
      end;
    end;
  end;
  Invalidategrids;
end; //TForm1.MoveNums;
```


В-третьих, кроме проверок, в этой же функции мы соединим приятное с полезным: запишем число групп в каждом ряду в массив, чтобы не пересчитывать их каждый раз, когда нам потребуется это знать.

В-четвёртых, из элементарной арифметики известно, что общее количество клеток в группах не зависит от того, будем ли мы их пересчитывать слева направо или сверху вниз. Поэтому суммы чисел в верхнем и в левом числовых полях должны быть *равны*. Если это не так, то задачу и решать не стоит. Правда, равенство этих сумм вовсе не гарантирует правильности условия задачи, то есть это условие необходимое, но не недостаточное. Например, если обе суммы равны нулю, то условие задачи (числа) просто отсутствует. Это положение может возникнуть, если вы не оцифровали свой рисунок, а хотите его решить (вообще-то его и решать не нужно, но вдруг вам захотелось «приколоться»). А вот более «хитрые» ошибки можно обнаружить, только уже решая задачу.

```
  //проверить данные задачи
  function Testing(): Boolean ;
  var
    sumL, sumT: integer;
    i, j, n: integer;
  begin
    Result:= False;
    //если задача не загружена, то нечего решать:
    if NameFig='temp' then
```

```
    begin
      WavError;
      application.MessageBox('Вы не загрузили задачу!',NAME_PROG, MB_OK);
      exit
    end;
    //нормализовать запись чисел:
    MoveNums;
    //записать число групп в каждом ряду в массив
    //и подсчитать сумму чисел -
    //верхнее числовое поле:
    sumT:= 0;
    for i:= 0 to POLE_WIDTH-1 do
    begin
      n:= 0;
      //считаем в каждом столбце:
      for j:= 0 to TOP_POLE_HEIGHT-1 do
      begin
        if masColsNum[i,j].sNum = '' then break; //- числа кончились
        inc(n);
        //записать соотв. число:
        masColsNum[i,j].Num := strtoint(masColsNum[i,j].sNum);
        sumT:= sumT + masColsNum[i,j].Num
      end;
      //записать число групп:
      masColsNum[i,-1].Num:= n;
    end;
    //левое числовое поле:
    sumL:= 0;
    for i:= 0 to POLE_HEIGHT-1 do
    begin
      n:= 0;
      //считаем в каждой строке:
      for j:= 0 to LEFT_POLE_WIDTH-1 do
      begin
        if masRowsNum[j,i].sNum = '' then break; //- числа кончились
        inc(n);
        //записать соотв. число:
        masRowsNum[j,i].Num := strtoint(masRowsNum[j,i].sNum);
        sumL:= sumL + masRowsNum[j,i].Num
      end;
      //записать число групп:
      masRowsNum[-1,i].Num:= n;
    end;
    //проверить суммы чисел:
    if sumT + sumL= 0 then begin
      WavError;
      application.MessageBox('Вы   не   оцифровали   рисунок!',NAME_PROG,
MB_OK);
      exit
    end;
    if sumT <> sumL then
    begin
      WavError;
      s:= 'Сумма чисел сверху (' +inttostr(sumT)+
      ')'#10#13'не равна сумме чисел слева (' +inttostr(sumL)+')!';
      application.MessageBox(PChar(s),NAME_PROG, MB_OK);
      exit;
```

```
    end;

    //если в ряду есть нулевая группа, то других быть не должно
    //верхнее числовое поле:
    for i:= 0 to POLE_WIDTH-1 do
      //проверяем каждый столбец:
      for j:= 0 to TOP_POLE_HEIGHT-1 do
      begin
        if masColsNum[i,j].sNum = '' then break; //- числа кончились
        if (masColsNum[i,j].sNum = '0') and (masColsNum[i,-1].Num>1)
        then begin
          WavError;
          s:='Неверные данные в столбце '+ inttostr(i+1)+'!';
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);
          exit
        end;
      end;
    //левое числовое поле:
    for i:= 0 to POLE_HEIGHT-1 do
    //проверяем каждую строку:
      for j:= 0 to LEFT_POLE_WIDTH-1 do
      begin
        if masRowsNum[j,i].sNum = '' then break; //- числа кончились
        if (masRowsNum[j,i].sNum = '0') and (masRowsNum[-1,i].Num>1)
        then begin
          WavError;
          s:='Неверные данные в строке '+ inttostr(i+1)+'!';
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);
          exit
        end;
      end;
    //всё нормально:
    Result:= True;
  end; // Testing
```


Если вы были дюже внимательны, то могли разглядеть в процедуре «инородное тело» - *WavError*. Оно внедрено сюда специально для тех, кто не равнодушен к звуковым галлюцинациям, иначе говоря, эффектам, которые преследуют каждого пользователя *Windows*, если он не выключил колонки. Но - каждый имеет право отравлять себе жизнь всеми доступными ему способами, в том числе и самыми изощрёнными, к коим можно смело отнести музицирование на компьютере. А готовится это зелье так. Объявите там же, где и функцию *Testing*, три *процедуры*:

```
    procedure WavError;
    procedure WavReadyLine;
    procedure WavPobeda;
```

Которые выглядят так:

```
//===================== ЗВУКОВЫЕ ЭФФЕКТЫ ==========================
//ОШИБКА!
procedure TForm1.WavError;
begin
```

```
  if not sbtSound.Down then exit;
  //воспроизвести звуковой эффект:
  sndPlaySound('wav\error.wav', SND_ASYNC or SND_FILENAME);
end;
//РЯД РЕШЁН!
procedure TForm1.WavReadyLine;
begin
  if not sbtSound.Down then exit;
  sndPlaySound('Wav\ReadyLine.wav', SND_ASYNC or SND_FILENAME);
end;
//ЗАДАЧА РЕШЕНА!
procedure TForm1.WavPobeda;
begin
  if not sbtSound.Down then exit;
  sndPlaySound('Wav\pobeda.wav', SND_ASYNC or SND_FILENAME);
end;
```


Все звуковые файлы хранятся в папке *Wav,* которая находится в той же папке (ох, уж эти папки!), что и выполняемый файл программы. Но вы можете засунуть эту музыку и куда подальше. Только не забудьте указать программе путь к ней, чтобы не вышла по вашему велению сказочная история, в которой герою предлагалось пойти туда не знаю куда.

Если вам этого звукового сопровождения (п)оказалось мало (а вдруг вы меломан), то вы можете добавить сколько угодно своих эффектов и расставить их в программе на каждом шагу. Тогда при решении задачи у вас не только глаза вылезут, но и ушки завернутся!

Но музыку, даже самую противную, любят не все, поэтому её услышит только тот, кто по неосторожности нажмёт кнопку *sbtSound* да так и оставит её без присмотра.

И вот условия задачи проверены, вся цифирь оказалась на месте, и нам уже не отвертеться от решения оной задачи. Сейчас мы сделаем много лишнего, что, однако, необходимо нам для отыскания общего алгоритма решения любых японских кроссвордов.

Давайте заставим компьютер выполнять все *Золотые правила*.

Они достаточно просты, и, конечно, каждый помнит их назубок.

1. Ряды, в которых есть *нулевые группы*, можно сразу закрасить *белым*. Процедура *TestingZero* с лёгкостью сделает то, с чего мы начинали решение задачи-примера. Она перебирает все ряды чисел в числовых полях и ищет в них *нули*. Найдя такой ряд, она отмечает его как *решённый* (статус такого ряда, как вы помните, *stGreen*), окрашивает все клетки ряда в *белый* цвет и, наконец, считает разгаданные клетки (их число хранится в переменной *ReadyCells*). Так как эти операции очень просты и хорошо известны, то вам легко будет понять их действие в программе. Потом вы

можете и не пользоваться этой процедурой, так как общий алгоритм решения задачи обойдётся и без неё. С другой стороны, она ничуть и не мешает.

```
//отмечаем пустые строки
  procedure TestingZero;
  var
    i, j, n: integer;
  begin
    //проверяем верхнее числовое поле:
    for i:= 0 to POLE_WIDTH-1 do
      //проверяем каждый столбец:
      for j:= 0 to masColsNum[i,-1].Num-1 do
      begin
        if (masColsNum[i,j].sNum = '0') then //- нашли
        begin
          {s:='Есть 0 в столбце '+ inttostr(i+1)+'!';
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
          //этот столбец решён:
          masColsNum[i,-1].StatusGroup:= stGreen;
          //закрашиваем столбец поля белым цветом:
          for n:= 0 to POLE_HEIGHT-1 do
          begin
            if masPole[i,n]=NUM_GRAY then //- эти клетки разгаданы?
            begin
              masPole[i,n]:= NUM_WHITE;
              inc(ReadyCells)
            end;
          end;
        end;
      end;
    //проверяем левое числовое поле:
    for i:= 0 to POLE_HEIGHT-1 do
    //проверяем каждую строку:
      for j:= 0 to masRowsNum[-1,i].num-1 do
      begin
        if (masRowsNum[j,i].sNum = '0')then //- ашли
        begin
          //эта строка решена:
          masRowsNum[-1,i].StatusGroup:= stGreen;
          {s:='Есть 0 в строке '+ inttostr(i+1)+'!';
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
          //закрашиваем строку поля белым цветом:
          for n:= 0 to POLE_WIDTH-1 do
          begin
            if masPole[n,i]=NUM_GRAY then //- эти клетки разгаданы?
            begin
              masPole[n,i]:= NUM_WHITE;
              inc(ReadyCells)
            end;
          end;
        end;
      end;
    InvalidateGrids;
  end; // TestingZero
```

2. По *Правилу 2*, ряды, в которых есть число, равное размеру поля, можно сразу закрасить *чёрным*. Кодируем *Правило 2* аналогично предыдущему. Единственное отличие – мы должны проверить, не «накладываются» ли чёрные клетки на белые (они могли появиться после выполнения *первого Правила*). Если это так, то в условии задачи имеются ошибки, решить её невозможно, а функция возвращает значение *False*.

```
//отмечаем полные строки
function TestingFullLine(): Boolean;
var
  i, n: integer;
begin
  Result:= TRUE;
  //проверяем верхнее числовое поле:
  for i:= 0 to POLE_WIDTH-1 do
    if masColsNum[i,0].Num = POLE_HEIGHT then //нашли
    begin
      //s:='Есть полный столбец '+ inttostr(i+1)+'!';
      //application.MessageBox(PChar(s),NAME_PROG, MB_OK);
      //закрашиваем столбец поля чёрным цветом:
      for n:= 0 to POLE_HEIGHT-1 do
      begin
        if masPole[i,n]<>NUM_GRAY then //- эти клетки уже разгаданы?
        begin
          s:='Неверные числа в столбце '+ inttostr(i+1)+#10#13+
          'и строке ' + inttostr(n+1) +'!';
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);
          Result:= FALSE;
          exit
        end
        else
        begin
          masPole[i,n]:= NUM_BLACK;
          inc(ReadyCells)
        end;
      end;
  //этот столбец решён:
  masColsNum[i,-1].StatusGroup:= stGreen;
  end;
  //проверяем левое числовое поле:
  for i:= 0 to POLE_HEIGHT-1 do
    if masRowsNum[0,i].Num = POLE_WIDTH then //- нашли
    begin
      //s:='Есть полная строка '+ inttostr(i+1)+'!';
      //application.MessageBox(PChar(s),NAME_PROG, MB_OK);
        //закрашиваем строку поля чёрным цветом:
        for n:= 0 to POLE_WIDTH-1 do
        begin
          if masPole[n,i]=NUM_WHITE then //- эта клетка белая?
          begin
            s:='Неверные числа в строке '+ inttostr(i+1)+#10#13+
            'и столбце ' + inttostr(n+1) +'!';
            application.MessageBox(PChar(s),NAME_PROG, MB_OK);
            Result:= FALSE;
            exit
          end
```

```
            else if masPole[n,i]=NUM_GRAY then//эта клетка серая?
            begin
              masPole[n,i]:= NUM_BLACK;
             inc(ReadyCells)
            end;
          end;
      //эта строка решена:
      masRowsNum[-1,i].StatusGroup:= stGreen;
      end;
    InvalidateGrids;
  end; // TestingFullLine
```


Судьбина у этой функции тоже горькая – как пример перевода правил с русского языка на дельфийский она ещё ничего себе, но программе совершенно не нужна.

3. По *Правилу 3*, мы отыскиваем ряды с *полными суммами* и закрашиваем клетки всех групп *чёрным*, а между ними по одной клетке – *белым*. При этом мы проверяем, не «накладываются» ли чёрные клетки на уже известные белые, и наоборот. Так как некоторые ряды уже могли быть решены раньше (тогда их *StatusGroup* = *stGreen*), то их мы пропускаем. Не забываем подсчитывать решённые клетки (их количество равно сумме *серых* клеток, которые были в разгаданном ряду).

Для удобства введём две *функции*. Объявите их в разделе *private*:

```
function GetNumGroup(Location: TPoleLocation; nLine: integer): in-
teger;
function GetLenGroup(Location: TPoleLocation; nLine, nGroup: inte-
ger): integer;
```

Для них пригодится новый *тип*:

```
//положение числового поля: слева, сверху
type TPoleLocation= (plLEFT, plTOP);
```

Первая функция возвращает число групп в заданном ряду верхнего или левого полей, *вторая* – длину заданной группы:

```
//ПОЛУЧИТЬ КОЛИЧЕСТВО ГРУПП В РЯДУ nLine левого (LEFT) или
//верхнего (TOP) числового поля
function TForm1.GetNumGroup(Location: TPoleLocation; nLine: integer): in-
teger;
begin
  case Location of
    plLEFT: Result:= masRowsNum[-1, nLine].Num;
    plTOP:  Result:= masColsNum[nLine, -1].Num;
    else Result:= -1;
  end;
end;
//ПОЛУЧИТЬ ДЛИНУ ГРУППЫ nGroup В РЯДУ nLine левого (LEFT) или
//верхнего (TOP) числового поля
function TForm1.GetLenGroup(Location: TPoleLocation; nLine, nGroup: inte-
ger)
```

```
         : integer;
begin
  case Location of
    plLEFT: Result:= masRowsNum[nGroup, nLine].Num;
    plTOP:  Result:= masColsNum[nLine, nGroup].Num;
    else Result:= -1;
  end;
end;
```

Так как это правило значительно сложнее предыдущих, то и код получится *подлин-нееееe*:


```
  //отмечаем строки с полными суммами
  function TestingFullSum(): Boolean;
  var
    i, j, n, k, sum: integer;
  begin
    Result:= TRUE;
    //проверяем верхнее числовое поле:
    for i:= 0 to POLE_WIDTH-1 do
      if masColsNum[i,-1].StatusGroup<> stGreen then //- столбец ещё не решён
      begin
        sum:= 0;
        //проверяем каждый столбец
        //находим сумму для каждого столбца:
        for j:= 0 to masColsNum[i,-1].Num-1 do
          sum:= sum + masColsNum[i,j].Num;
        sum:= sum + masColsNum[i,-1].Num-1;
        if sum= POLE_HEIGHT then //- нашли
        begin
          {s:='Есть полная сумма в столбце '+ inttostr(i+1)+'!';
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
          //этот столбец решён:
          masColsNum[i,-1].StatusGroup:= stGreen;
          //закрашиваем столбец поля:
          k:= 0;
          for n:= 0 to masColsNum[i,-1].Num-1 do //-по всем группам
          begin
            //закрасить группу чёрным:
            for j:= k to k+ masColsNum[i,n].Num-1 do
            begin
              if masPole[i,j]= NUM_WHITE then //эта клетка белая?
              begin
                s:='Неверные числа в столбце '+ inttostr(i+1)+#10#13+
                'и строке ' + inttostr(j+1) +'!';
                application.MessageBox(PChar(s),NAME_PROG, MB_OK);
                Result:= FALSE;
                exit
              end
              else if masPole[i,j]=NUM_GRAY then//- эта клетка серая?
              begin
                masPole[i,j]:= NUM_BLACK;
                inc(ReadyCells)
              end;
            end; // for j
            //поставить белую клетку между группами:
            if n < GetNumGroup(plTOP, i)-1 then
            begin
              k:= k+ masColsNum[i,n].Num;
              if masPole[i,k]= NUM_BLACK then //эта клетка чёрная?
              begin
```

```
              s:='Неверные числа в столбце '+ inttostr(i+1)+#10#13+
              'и строке ' + inttostr(k+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
              Result:= FALSE;
              exit
            end
            else if masPole[i,k]=NUM_GRAY then//- эта клетка серая?
            begin
              masPole[i,k]:= NUM_WHITE;
              inc(ReadyCells);
            end;
            inc(k);
          end; //n < GetNumGroup(plTOP, i)-1
        end; // for
      end; //if sum= POLE_HEIGHT
    end; //if masColsNum[i,-1].StatusGroup<> stGreen

  //проверяем левое числовое поле:
  for i:= 0 to POLE_HEIGHT-1 do
    if masRowsNum[-1,i].StatusGroup<> stGreen then //- строка ещё не решена
    begin
      sum:= 0;
      //проверяем каждый строку
      //находим сумму для каждой строки:
      for j:= 0 to masRowsNum[-1,i].Num-1 do
        sum:= sum + masRowsNum[j,i].Num;
      sum:= sum + masRowsNum[-1,i].Num-1;
      if sum= POLE_WIDTH then //нашли
      begin
        {s:='Есть полная сумма в строке '+ inttostr(i+1)+'!';
        application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
        //эта строка решена:
        masRowsNum[-1,i].StatusGroup:= stGreen;
        //закрашиваем строку поля:
        k:= 0;
        for n:= 0 to masRowsNum[-1,i].Num-1 do //- по всем группам
        begin
          //закрасить группу чёрным:
          for j:= k to k+ masRowsNum[n,i].Num-1 do
          begin
            if masPole[j,i]= NUM_WHITE then //- эта клетка белая?
            begin
              s:='Неверные числа в строке '+ inttostr(i+1)+#10#13+
              'и столбце ' + inttostr(j+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
              Result:= FALSE;
              exit
            end
            else if masPole[j,i]=NUM_GRAY then//- эта клетка серая?
            begin
              masPole[j,i]:= NUM_BLACK;
              inc(ReadyCells)
            end;
          end; // for j
          //поставить белую клетку между группами:
          if n< GetNumGroup(plLEFT, i)-1 then
          begin
            k:= k+ masRowsNum[n,i].Num;
            if masPole[k,i]= NUM_BLACK then //- эта клетка чёрная?
            begin
              s:='Неверные числа в строке '+ inttostr(i+1)+#10#13+
              'и  столбце' + inttostr(k+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
```

```
              Result:= FALSE;
              exit
            end
            else if masPole[k,i]=NUM_GRAY then//- эта клетка серая?
            begin
              masPole[k,i]:= NUM_WHITE;
              inc(ReadyCells);
            end;
            inc(k);
          end; //n< GetNumGroup(plLEFT, i)-1
        end; // for
      end; //if sum= POLE_HEIGHT
    end; //if masRowsNum[i,-1].StatusGroup<> stGreen
    InvalidateGrids;
end; //TestingFullSum
```


Эту функцию желательно оставить в программе, она может пригодиться в начале решения задачи. Но программа справится с любым кроссвордом и без неё.

4. По *Правилу 4*, если единственная группа в ряду длиннее половины ряда, то часть ряда можно закрасить *чёрным*. Функция *TestingLongLine* последовательно перебирает все ряды, и, если находит неразгаданный ряд и в нём *единственное* число, большее половины длины соответствующего ряда, то закрашивает часть клеток *чёрным* цветом. Как определить эти клетки, мы уже подробно рассматривали раньше.

```
//отмечаем длинные строки
function TestingLongLine(): Boolean;
var
  i, j, sum: integer;
begin
  Result:= TRUE;
  //проверяем верхнее числовое поле:
  for i:= 0 to POLE_WIDTH-1 do
    if masColsNum[i,-1].StatusGroup<> stGreen then //- столбец ещё не решён
    begin
      //в ряду должна быть одна группа:
      if GetNumGroup(plTOP, i)=1 then
      begin
        sum:= GetLenGroup(plTOP, i, 0)*2;
        if sum - POLE_HEIGHT >0 then //нашли
        begin
          {s:='Длинный столбец '+ inttostr(i+1)+'!';
          s:= s+'beg= '+inttostr((POLE_HEIGHT-GetLenGroup(plTOP, i, 0)));
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
          //закрашиваем часть столбца поля чёрным:
          for j:= (POLE_HEIGHT- GetLenGroup(plTOP, i, 0))
                                      to GetLenGroup(plTOP, i, 0)-1 do
          begin
            if masPole[i,j]= NUM_WHITE then //- эта клетка белая?
            begin
              s:='Неверные числа в столбце '+ inttostr(i+1)+#10#13+
              'и строке ' + inttostr(j+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
              Result:= FALSE;
              exit
            end
            else if masPole[i,j]=NUM_GRAY then//- эта клетка серая?
            begin
              masPole[i,j]:= NUM_BLACK;
```

```
              inc(ReadyCells)
            end;
          end; // for j
        end; // sum - POLE_HEIGHT >0
      end; // if GetNumGroup(TOP, i)=1
    end; //if masColsNum[i,-1].StatusGroup<> stGreen
  //проверяем левое числовое поле:
  for i:= 0 to POLE_HEIGHT-1 do
    if masRowsNum[-1,i].StatusGroup<> stGreen then //- столбец ещё не решён
    begin
      //в ряду должна быть одна группа:
      if GetNumGroup(plLEFT, i)=1 then
      begin
        sum:= GetLenGroup(plLEFT, i, 0)*2;
        if sum - POLE_WIDTH >0 then //нашли
        begin
          {s:='Длинная строка '+ inttostr(i+1)+'!';
          s:= s+'beg= '+inttostr((POLE_WIDTH-GetLenGroup(plLEFT, i, 0)));
          application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
          //закрашиваем часть строки поля чёрным:
          for j:= (POLE_WIDTH- GetLenGroup(plLEFT, i, 0))
                                    to GetLenGroup(plLEFT, i, 0)-1 do
          begin
            if masPole[j,i]= NUM_WHITE then //эта клетка белая?
            begin
              s:='Неверные числа в строке '+ inttostr(i+1)+#10#13+
              'и столбце ' + inttostr(j+1) +'!';
              application.MessageBox(PChar(s),NAME_PROG, MB_OK);
              Result:= FALSE;
              exit
            end
            else if masPole[j,i]=NUM_GRAY then//- эта клетка серая?
            begin
              masPole[j,i]:= NUM_BLACK;
              inc(ReadyCells)
            end;
          end; // for j
        end; // sum - POLE_WIDTH >0
      end; // if GetNumGroup(TOP, i)=1
    end; //if masRowsNum[i,-1].StatusGroup<> stGreen
   InvalidateGrids;
end; // TestingLongLine
```

Безусловно, эта функция самая «прогрессивная» из всей четвёрки, но и она – только шаг к главному алгоритму. Теперь мы знаем, как перевести «человеческие» правила на компьютерный язык. Можно было бы продолжить наши упражнения дальше и научить компьютер пользоваться *Железными правилами*, но мы этого делать не станем, а лучше подумаем, что общего у всех *Правил* и как этим воспользоваться.

Мандрагора, или Зри в корень!

Возьмите чистую посуду…

Из анекдота про мужика, опрометчиво взявшегося за стряпню

Энэ бэнэ рес!
Квинтер финтер жес!
Энэ бэнэ ряба,
Квинтер финтер жаба…

Незнайкина считалочка

…Это первое. Второе…

Ельциновская риторика

Главную идею алгоритма решения японских кроссвордов можно вывести непосредственно из *Правил*: *если при всех возможных расстановках групп в ряду какие-либо клетки* ***всегда*** *оказываются чёрными (или белыми), то они таковыми и являются на самом деле.* Например, *Правила 1-3* потому и верны, что клетки можно расставить *единственным* способом. *Правило 4* легко проверить, если выписать все расстановки групп в ряду. Но для одной группы всё настолько очевидно, что достаточно посмотреть положение группы в крайних положениях (об этом было рассказано достаточно подробно). Случай с двумя группами в ряду также обсуждался, но он уже не столь очевиден. А в реальных задачах число групп доходит до десятка и более в одном ряду. Проверить все расстановки вручную очень трудно, поэтому приходится «изобретать» и другие правила, которыми проще пользоваться при *ручном* решении японских головоломок. Компьютер же, напротив, «любит» не думать, а перебирать варианты расстановок. Причём делает он это настолько быстро, что ему вообще не нужны никакие правила.

Итак, первое, чему мы должны научить компьютер, - это находить все расстановки групп в ряду. Этот комбинаторный *алгоритм* очень прост:

1. Размещаем все группы *чёрных* клеток в их крайнем левом положении, а все остальные клетки ряда окрашиваем в *белый* цвет (в программе это делается иначе: сначала все клетки красим белым, а потом ставим чёрные группы; результат, естественно, от *перемены мест слагаемых* не изменится). Так мы получаем *первую* расстановку.

2. Пока это возможно, сдвигаем *последнюю группу* вправо на одну клетку. Каждый сдвиг даёт новую расстановку.

3. Если это возможно, сдвигаем *предыдущую группу* на одну клетку вправо, а все следующие выставляем через одну белую клетку за ней. Получаем ещё один вариант расстановки и переходим к *пункту 2*. Иначе - повторяем это действие.

Проиллюстрируем действие алгоритма на *примере*. Пусть длина ряда равняется 10 клеткам. В нём содержится 3 группы чёрных клеток, длиной 3, 2 и 1 клетка, соответственно. Тогда все расстановки групп легко получить, следуя алгоритму (Рис. 3.35).

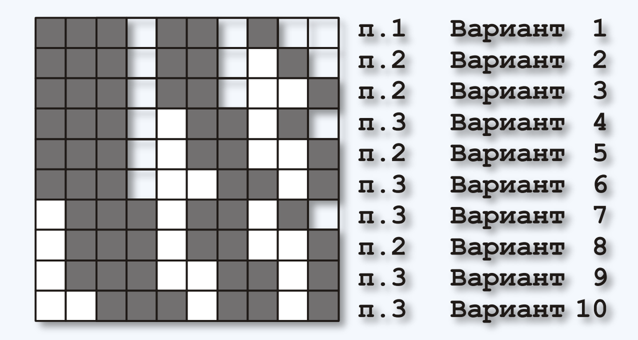


Рис. 3.35. Черно-белая комбинаторика

Легко заметить, что одна клетка ряда (третья слева) при *всех* расстановках групп *чёрная*, поэтому мы можем однозначно утверждать, что она именно чёрная. Цвет остальных клеток пока определить нельзя.

Так как не разгаданные клетки мы обозначаем *серым* цветом, то итоговый (или результирующий) ряд должен быть таким (Рис. 3.36).

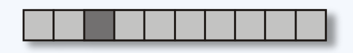


Рис. 3.36. Одна клетка ряда решена

То есть одну клетку ряда мы смело можем (и должны!) выкрасить в *чёрный* цвет.

В процедуре *TestingOneLine*, реализующей описанный алгоритм, чтобы не запоминать *все* расстановки групп (их может оказаться очень много), сначала за итоговый ряд принимается *первая* расстановка (вариант), а затем каждая следующая (текущая) сравнивается с ней: если цвет клеток в одинаковых позициях у итоговой и текущей расстановок *не совпадают*, то клетка не может быть разгадана, то есть в итоговом ряду она *серая*, иначе

цвет клетки в итоговом ряду не изменяется. Проверьте – результат окажется в точности такой же, что и в примере.

Недостаток нашего алгоритма в том, что он не учитывает уже разгаданные клетки в ряду. Допустим, что в разобранном примере разгадана начальная клетка – она *белая*. Тогда первые 6 расстановок групп можно проигнорировать, ведь в них начальная клетка *чёрная*. Отсюда следует *вывод*: для каждой расстановки нужно проверять, совместима ли она с уже известными клетками ряда. Если *чёрная* клетка текущей расстановки «накладывается» на *белую* клетку исходного ряда (того, что находится на игровом поле) или, наоборот, *белая* клетка накладывается на *чёрную*, то этот вариант расстановки не годится.

То же, но в картинках (Рис. 3.37).

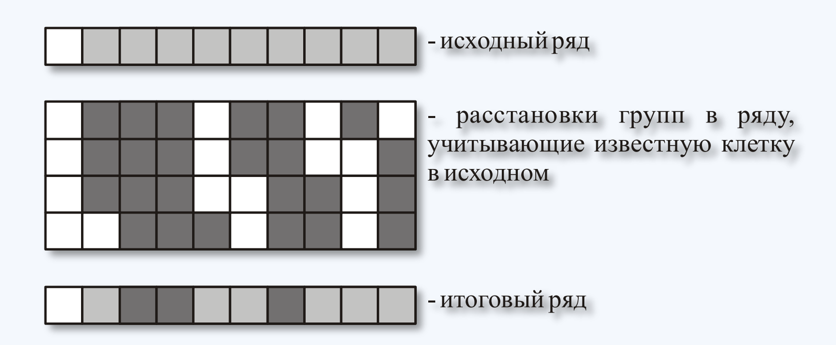


Рис. 3.37. Избегайте накладок!

Из этих рисунков видно, что даже одна известная клетка в исходном ряду позволяет отгадать значительно больше клеток, чем в совершенно «нетронутом», а число вариантов размещения групп сократилось до четырёх.

Если окажется, что разместить группы в ряду можно *единственным* способом (*nVar=1*), то найдено решение для *всего* ряда. Если же *ни одна* из расстановок групп (*nVar= 0*) не совместима с уже разгаданными клетками, то этот ряд не может быть решён. Такое положение возникает, когда в условии задачи имеется ошибка или ранее мы сделали неверное предположение о цвете одной из клеток (об этом мы поговорим немного погодя).

Функция TestingOneLine как раз и возвращает число возможных вариантов расстановки групп в ряду:

```
//------ ищем подходящие комбинации клеток в ряду --------
function TestingOneLine(nGroup: integer;
                        LenGroup: array of integer;
                        LenLine: integer;
                        masSource: array of integer;
```

```
                        var masResult: array of integer): integer;
//функция возвращает количество возможных расстановок групп в заданном ряду
//nGroup: integer;                              // число (0..) групп в ряду
//LenGroup: array[0..MAX_CELLNUM-1] of integer; //длина группы
//LenLine: integer;                             //длина ряда
//masSource: array[0..100] of integer;          //исходная полоска
//masResult: array[0..100] of integer;          //итоговая полоска
label MoveLastGroup, NextMove, PredMove;
var
  ptr: integer; // номер (0..nGroup-1) сдвигаемой группы
  pos: array[0..MAX_CELLNUM-1] of integer; //позиция первой клетки группы
  nVar: integer;                      //число найденых вариантов полосок
  masVar: array[0..100] of integer; //очередная полоска
  i: integer;
  //проверить, может ли группа разместиться в ряду
  function ExaminePos(n: integer): Boolean;
  //n - номер (0..) группы
  begin
    Result:= FALSE;
    if pos[n]+LenGroup[n]<=LenLine then Result:= TRUE;
  end;
  //записать расстановку:
  function WriteVariant(): Boolean;
  var i, j: integer;
  begin
    Result:= TRUE;
    //выставить белые клетки по длине ряда:
    for i:= 0 to LenLine-1 do masVar[i]:= NUM_WHITE;
    //расставить все группы чёрных клеток:
    for i:= 0 to nGroup-1 do
      for j:= pos[i] to pos[i]+LenGroup[i]-1 do
        masVar[j]:= NUM_BLACK;

    //скопировать первую расстановку в итоговую полоску:
    if nVar= 1 then
     for i:= 0 to LenLine-1 do masResult[i]:= masVar[i];
    //проверить, ложится ли полоска на уже имеющуюся на поле -
    //чёрная клетка текущей полоски не должна накладываться на
    //белую клетку поля, а белая - на чёрную:
    for i:= 0 to LenLine-1 do
      if (masVar[i]<> masSource[i]) and (masSource[i]<> NUM_GRAY) then
      begin
        Result:= False; exit;//- не ложится!
      end;
    //скорректировать итоговую полоску - если цвета клеток итоговой
    //и текущей полосок в одинаковых позициях разные, то
    //цвет клетки серый, иначе - без изменений
    //сравниваем полоски по всей длине:
    for i:= 0 to LenLine-1 do
      if (masVar[i]<> masResult[i]) then masResult[i]:= NUM_GRAY;
  end; // WriteVariant

begin
  //номера начальных клеток полосок записываем
  //в массив pos:
  pos[0]:= 0;
  For i:= 1 to nGroup-1 do pos[i]:= pos[i-1]+ LenGroup[i-1]+1;
  //начальная расстановка групп:
  nVar:=1;
  if WriteVariant= FALSE then nVar:= 0;
//начинаем сдвигать последнюю группу -
MoveLastGroup:
  ptr:= nGroup-1;
```

```
  //передвигаем группу в след. клетку -
  NextMove:
    Inc(pos[ptr]);
    if ExaminePos(ptr) then //можно сдвигать
    begin
      inc(nVar);
      //записать расстановку:
      if WriteVariant= FALSE then dec(nVar);
      goto NextMove; //- сдвигаем до края
    end;
    //сдвигать нельзя -->
//переходим к предыдущей группе:
PredMove:
    dec (ptr);
    if ptr<0 then //- все группы сдвинуты до края
    begin
      {s:='Все варианты: '+ inttostr(nVar);
      application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
      Result:= nVar;
      exit
    end;
    //ищем дальше -->
    //перемещаем начало текущей группы:
    inc(pos[ptr]);
    //проверяем:
    if ExaminePos(ptr)= FALSE then
    begin
      {s:='Все варианты: '+ inttostr(nVar);
      application.MessageBox(PChar(s),NAME_PROG, MB_OK);}
      Result:= nVar;
      exit
    end;
    //расставляем следующие группы сразу после текущей:
    For i:= ptr+1 to nGroup-1 do
    begin
      pos[i]:= pos[i-1]+LenGroup[i-1]+1;
      //проверить:
      if ExaminePos(i)= FALSE then //- для след. групп не хватает места
        goto PredMove;
    end;
    //всё нормально - записать новую расстановку:
      inc(nVar);
      if WriteVariant= FALSE then dec(nVar);
    //ищем след. растановку:
    Goto MoveLastGroup;
  end; // function TestingOneLine()
```

Эта функция обрабатывает один ряд поля (его копия передаётся в массиве *masSource*). Рассмотрим работу ещё одной функции – *TestingBrain*. Для каждого ещё не разгаданного ряда она вызывает *TestingOneLine*, а затем анализирует её значение. Если оно равно *0*, то *TestingBrain* возвращает значение *False* (не разгадана ни одна клетка), а *NoVar* устанавливает в *True* (найден нерешаемый ряд). Если значение равно *1*, то *TestingBrain* возвращает значение *True*, записывает разгаданные клетки ряда в массив *masPole* и подсчитывает число известных клеток:

```
  //проверить, можно ли ещё поставить белые и чёрные клетки в ряд
  function TestingBrain(): Boolean;
```

```
//= TRUE, если разгадана хотя бы одна клетка
var
  i, j, n: integer;
  ng: integer;
  LG: array[0..MAX_CELLNUM-1] of integer; //длина групп
  masSrc: array[0..100] of integer;       //исходная полоска
  masResult: array[0..100] of integer;    //итоговая полоска
begin
  Result:= FALSE;

  //проверяем верхнее числовое поле:
  for i:= 0 to POLE_WIDTH-1 do //- по ширине поля
    //если столбец ещё не решён -->
    if masColsNum[i,-1].StatusGroup<> stGreen then
    begin
      //число групп в ряду:
      ng:= GetNumGroup(plTOP, i);
      //длина групп в ряду:
      for j:= 0 to ng-1 do lg[j]:= masColsNum[i,j].num;
      //скопировать ряд поля -> исходная полоска:
      for j:= 0 to POLE_HEIGHT-1 do masSrc[j]:= masPole[i,j];
      //получить новую полоску (ряд) на поле:
      n:= TestingOneLine(ng, lg, POLE_HEIGHT, masSrc, masResult);
      //s:='n= ' + inttostr(n)+ ' (Столбец: '+ inttostr(i);
      //application.MessageBox(PChar(s),NAME_PROG, MB_OK);
      //если нет ни одной подходящей растановки групп в ряду -->
      if n=0 then begin
        Result:= FALSE;
        NoVar:= TRUE;
        exit
      end;
      //если имеется единственная расстановка групп в ряду -->
      //ряд решён:
      if n= 1 then masColsNum[i,-1].StatusGroup:= stGreen;
      //вывести итоговую полоску на поле:
      for j:= 0 to POLE_HEIGHT-1 do
      begin
        if (masPole[i,j]= NUM_GRAY) and (masResult[j]<> NUM_GRAY) then
        begin
          Result:= TRUE;
          //ещё 1 клетка разгадана:
          inc(ReadyCells);
          masPole[i,j]:= masResult[j];
        end
      end;
      InvalidateGrids;
    end; // if masColsNum[i,-1].StatusGroup<> stGreen

  //проверяем левое числовое поле:
  for i:= 0 to POLE_HEIGHT-1 do
    //- столбец ещё не решён ->
    if masRowsNum[-1,i].StatusGroup<> stGreen then
    begin
      //число групп в ряду:
      ng:= GetNumGroup(plLEFT, i);
      for j:= 0 to ng-1 do lg[j]:= masRowsNum[j,i].num;
      for j:= 0 to POLE_WIDTH-1 do masSrc[j]:= masPole[j,i];
      n:= TestingOneLine(ng, lg, POLE_WIDTH, masSrc, masResult);
      //если нет ни одной подходящей растановки групп в ряду -->
      if n=0 then begin
        Result:= FALSE;
        NoVar:= TRUE;
        exit
```

```
        end;
        //если имеется единственная расстановка групп в ряду -->
        //ряд решён:
        if n= 1 then masRowsNum[-1,i].StatusGroup:= stGreen;
        //вывести итоговую полоску на поле:
        for j:= 0 to POLE_WIDTH-1 do
        begin
          if (masPole[j,i]= NUM_GRAY) and (masResult[j]<> NUM_GRAY) then
            begin
              Result:= TRUE;
              //ещё 1 клетка разгадана:
              inc(ReadyCells);
              masPole[j,i]:= masResult[j];
            end
        end;
        InvalidateGrids;
      end; // if masRowsNum[-1,i].StatusGroup<> stGreen
  end; //function TestingBrain()
```


Если отвлечься от частностей, то алгоритм решения японских кроссвордов получился очень простым. Конечно, для ручного гадания он непригоден, так как придётся перебирать большое число расстановок групп, а вот для машинного – *то что нужно*! Так как здесь проводится *полный* перебор размещений групп, то любая задача будет неизбежно решена (или будет установлено, что она решений не имеет). «Слабым местом» алгоритма является генерация *всех* размещений групп в ряду, даже если известно точное положение некоторых из них (разгаданные группы). Это может привести к тому, что задачи, имеющие длинные ряды и большое количество очень коротких групп (1-2 клетки), будут решаться несколько минут (обычно на это уходят секунды, причём и это время можно сильно сократить, если убрать перерисовку полей во время решения задачи, правда, тогда вам уже не удастся насладиться самим процессом решения). Впрочем, это не столько недостаток алгоритма, сколько изъян в самой задаче: хорошо сработанный кроссворд, как правило, имеет в рядах не более 5-7 групп, поэтому ответ будет найден очень быстро. В откровенно слабых заданиях встречаются ряды с десятком и более одноклеточных групп. Сделаны они наверняка не для компьютера, и нетрудно себе представить адские муки разгадчика, часами отыскивающего места для этих групп, чтобы в итоге получить банального ёжика, раскрашенного, как шахматная доска. Если вы планируете самостоятельно составлять японские кроссворды, *будьте добрее к людям, не мучайте их понапрасну*!

Как я уже отмечал, этот алгоритм с лихвой перекрывает все написанные и не написанные ещё правила решения кроссвордов, поэтому в программе можно оставить только его, а функции, реализующие *Правила*, просто удалить. И хотя в этом нет никакой необходимости, но вы можете попытаться *улучшить* предложенный алгоритм. Например, если число *серых* клеток в итоговом ряду сравняется с числом таких клеток в исходном ряду, то нет

смысла дальше перебирать варианты размещения групп – ни одну клетку ряда в этот раз разгадать не удастся.

И нам осталось рассмотреть автоматическое решение задач в целом, то есть узнать, что происходит в программе после нажатия на кнопку *sbtStart*.

Магическая кнопка, или Первые аплодисменты

Кто ищет, тот всегда найдёт.

Из оптимистической песни

В первую очередь, нужно изменить режим работы программы - *SetStatus('ПОИСК');* Теперь все кнопки будут «знать», что программа занята важным делом, и не смогут ей помешать. Для удобства мы напишем небольшую *процедуру*:

```
//УСТАНОВИТЬ СТАТУС ПРОГРАММЫ
procedure TForm1.SetStatus (s: string);
begin
  status:= s;
  StatusBar1.Panels[5].text := s;
  application.ProcessMessages;
end;
```

Не забудьте объявить её в разделе *private*:

```
procedure SetStatus (s: string);
```

Затем мы проверяем правильность данных задачи и подсчитываем число уже разгаданных клеток. Поначалу кажется, что этого и делать не нужно, ведь задачу мы ещё не решали, но не торопитесь – вы могли начать решение вручную, а потом, отчаявшись, предоставили свою судьбу машине.

Часть клеток можно разгадать, воспользовавшись *Правилами*, реализованными в рассмотренных ранее функциях.

На этом лирическая часть решения завершается, и начинаются трудовые будни функции *TestingBrain*. Если она «опознала» хотя бы одну клетку, значит, ситуация на поле изменилась, и мы можем попытаться продвинуться дальше в своём решении, опять же вызывая *TestingBrain*. Так как решение может и затянуться (или вы просто передумаете решать кроссворд), то следует позаботиться о выходе из этой круговерти событий. Са-

мый правильный способ – нажать кнопку *sbtStop*, которая установит флаг выхода из процедуры. Ообъявите *переменную* так же, как и раньше:

```
flgExit: boolean= False;
```

А уж если вы очень торопитесь, то просто закройте программу.

```
//ОСТАНОВИТЬ РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.sbtStopClick(Sender: TObject);
begin
  if status<>'ПОИСК' then exit;
  flgExit:=true
end;

//ЗАКРЫТЬ ПРОГРАММУ
procedure  TForm1.FormClose(Sender:  TObject;  var  Action:  TCloseAc-
tion);
begin
  flgExit:= true;
end;
```


Гоняя функцию *TestingBrain* по кругу, мы должны на каждом «витке» проверять, не разгадала ли она часом уже *все* клетки. Рано или поздно так и случится, и вот тут-то вы и увидите, что часть рядов «не решена», то есть не закрашена *зелёным* цветом. Это свидетельствует о том, что некоторые числа вообще не нужны для решения задачи. Но раз уж они есть, мы должны убедиться, что они «правильные». Решая чужую задачу, вы сможете позлорадствовать над горе-составителем, если найдёте ошибки, а, проверяя свою собственную, избежите участи предыдущего товарища.

И вот он – миг победы: всё сошлось, и задача покорно пала к вашим ногам. Если вы успели нажать кнопку *sbtSound*, то услышите в свою честь аплодисменты (не смущайтесь – вы их заслужили!).

Этим приятным звуковым эффектом заканчивается решение всех задач (конечно, «непорочных»), да не всегда так скоро. Я уже предупреждал вас о коварных составителях неудобоваримых задач, решение которых не обходится без русского авось, или – по-научному – тыка. Если *TestingBrain* вернётся ни с чем (найдёт коса на камень, и ни одна клетка не будет разгадана), то нам ничего другого не останется, как задать цвет какой-нибудь клетки насильно, а затем продолжить решение. Если ситуация повторится, придётся сделать ещё одно предположение и так далее (разумно ограничить число предположений, чтобы не тратить время на неудачные задачи. В этой программе предельное число предположений равно 20. Объявите константу *MAX_LEVEL*:

```
MAX_LEVEL= 20; //макс. сложность задачи (число предположений))
```

Текущий *уровень сложности* задачи хранится в переменной *Level*:

```
Level: Integer=0; //уровень сложности задачи (число предположений)
```

Но как выбрать сомнительную клетку, и в какой цвет её закрасить? – Мы поступим просто – первую попавшуюся от начала поля *серую* клетку сделаем *чёрной* (если предположение окажется неверным, то перекрасим её в *белый* цвет – других вариантов нет). Любители сложных задач могут поискать и более эффективный алгоритм подобных предположений. Другой вопрос: как мы сможем найти сомнительную клетку, если с её цветом не угадали, и как восстановить позицию на поле в тот судьбоносный момент? – Нужно запомнить всю необходимую информацию в массиве *SaveLevel* (объявите его там же, где и все переменные):


```
  //массив, в котором хранятся данные предыдущих уровней:
  SaveLevel: array[0..MAX_LEVEL] of TSaveLevel;
```

А хранить в нём мы будем вот что:

```
//данные уровня
type TSaveLevel = Record
  //статус рядов верхнего числового поля:
  TopStatus: array[0..MAX_POLE_WIDTH-1]of TStatusGroup;
  //статус рядов левого числового поля:
  LeftStatus: array[0..MAX_POLE_HEIGHT-1]of TStatusGroup;
  //копия поля:
  masPole: TPole;
  //коорд. "сомнительной" клетки
  XCell: integer;
  YCell: integer;
  //число разгаданных клеток:
  ReadyCells: integer;
end;
```

Этого вполне достаточно, чтобы безболезненно вернуться в случае надобности к нашим ошибкам и взяться за новые.

Но, даже решив задачу, в которой делались предположения, мы не можем быть уверены, что решили её правильно. То есть решение, безусловно, будет удовлетворять условиям задачи, но картинка получится неверной.

Чтобы никого не обидеть, в качестве примера дуалей приведу тестовую задачу (Рис. 3.38).

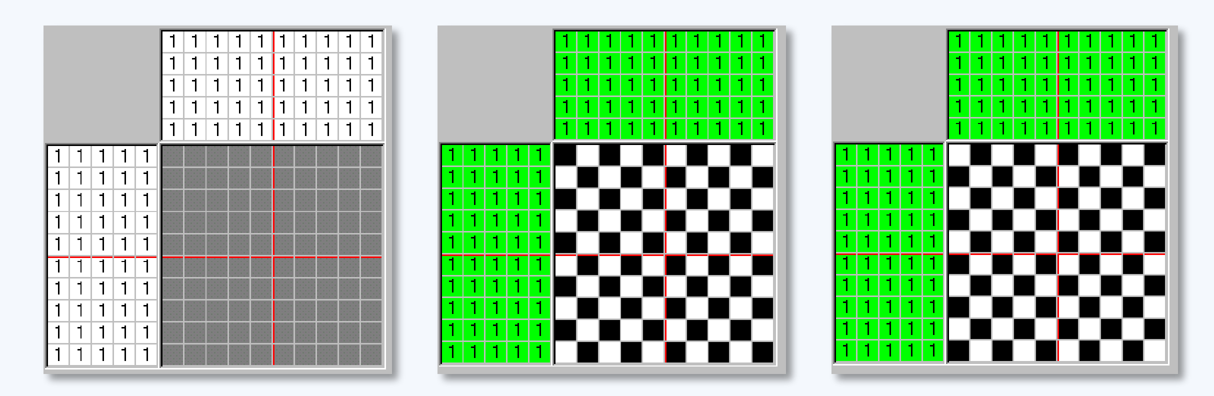


Рис. 3.38. И так, и сяк, и в клеточку!

Вы легко можете убедиться, что оба решения *верные*, хотя автор мог иметь в виду только одно из них. Мы не должны винить в этом нашу программу – она всё сделала правильно, а вся ответственность ложится исключительно на составителей. Задачи, имеющие побочные решения, это плохие задачи, и характеризуют автора не с лучшей стороны, поэтому обязательно проверяйте свои кроссворды, чтобы избежать досадных ляпсусов.

В процедуре, обрабатывающей нажатие кнопки *sbtStart*, текст некоторых функций заменён многоточием, так как он уже приводился полностью раньше. Что касается работы процедуры, то мы разобрали её достаточно подробно, чтобы у вас не возникало сомнений.

```
//=================================================
//=============== РЕШИТЬ ЗАДАЧУ =================
//=================================================
procedure TForm1.sbtStartClick(Sender: TObject);
label again;
var
  i, j: integer;
  s: string;
  NoVar: boolean;
  numVar: integer;   //число найденных вариантов решения задачи
  maxLevel: integer; //сложность решения задачи

  //проверить данные задачи
  function Testing(): Boolean ;
  . . .
  end; // Testing

  //проверить, не все ли клетки разгаданы:
  function IsReady(): Boolean;
  begin
    lblReady.Caption:= inttostr(ReadyCells);
    Result:= False;
    if ReadyCells= AllCells then Result:= TRUE; //готово!
  end;
```

```
//отмечаем пустые строки
procedure TestingZero;
. . .
end; // TestingZero

{//отмечаем полные строки
function TestingFullLine(): Boolean;
. . .
end; // TestingFullLine}

//отмечаем строки с полными суммами
function TestingFullSum(): Boolean;
. . .
end; //TestingFullSum

//отмечаем длинные строки
function TestingLongLine(): Boolean;
. . .
end; // TestingLongLine

//------ ищем подходящие комбинации клеток в ряду --------
function TestingOneLine(nGroup: integer;
                        LenGroup: array of integer;
                        LenLine: integer;
                        masSource: array of integer;
                        var masResult: array of integer): integer;
. . .
end; //function TestingBrain()

//сохранить данные заданного уровня
procedure SaveDataLevel(n: integer);
//n - номер уровня (0..)
var i, j: integer;
begin
  //сохранить статус рядов верхнего числового поля:
  for i:= 0 to POLE_WIDTH-1 do
    SaveLevel[n].TopStatus[i]:= masColsNum[i,-1].StatusGroup;
  //сохранить статус рядов левого числового поля:
  for i:= 0 to POLE_WIDTH-1 do
    SaveLevel[n].LeftStatus[i]:= masRowsNum[-1,i].StatusGroup;
  //сохранить копию поля:
  SaveLevel[n].masPole:= masPole;
  //заменить первую серую клетку на чёрную:
  inc(ReadyCells);
  //сохранить число разгаданных клеток:
  SaveLevel[n].ReadyCells:= ReadyCells;
  for j:= 0 to POLE_HEIGHT-1 do
  for i:= 0 to POLE_WIDTH-1 do
    if masPole[i,j]= NUM_GRAY then begin
      masPole[i,j]:= NUM_BLACK;
      //сохранить координаты "сомнительной" клетки:
      SaveLevel[n].XCell:= i;
      SaveLevel[n].YCell:= j;
      exit
    end;
end;  //SaveDataLevel

//загрузить данные заданного уровня
procedure LoadDataLevel(n: integer);
//n - номер уровня (0..)
var i: integer;
begin
```

```
    //загрузить статус рядов верхнего числового поля:
    for i:= 0 to POLE_WIDTH-1 do
      masColsNum[i,-1].StatusGroup:= SaveLevel[n].TopStatus[i];
    //загрузить статус рядов левого числового поля:
    for i:= 0 to POLE_WIDTH-1 do
      masRowsNum[-1,i].StatusGroup:= SaveLevel[n].LeftStatus[i];
    //загрузить копию поля:
    masPole:= SaveLevel[n].masPole;
    //заменить чёрную клетку на белую:
    masPole[SaveLevel[n].XCell,SaveLevel[n].YCell]:= NUM_WHITE;
    //загрузит число разгаданных клеток:
    ReadyCells:= SaveLevel[n].ReadyCells;
  end;  //LoadDataLevel

//
//===================== Р Е Ш А Е М  З А Д А Ч У =======================
//
begin
  if status='ПОИСК' then exit; //- задача уже решается
  SetStatus('ПОИСК');
  //выключить режимы перемещения и рисования при решении задачи:
  sbtDraw.Down:= FALSE;
  sbtMove.Down:= FALSE;

  numVar:=0;
  maxLevel:= 0;

  //проверить данные фигуры:
  if Testing=FALSE then begin SetStatus('ОЖИДАНИЕ'); exit end;// - ошибка!

  //ни одна клетка пока не разгадана:
  ReadyCells:= 0;
  //подсчитать число уже разгаданных клеток:
  for j:= 0 to POLE_HEIGHT-1 do
    for i:= 0 to POLE_WIDTH-1 do
      if masPole[i,j]<>NUM_GRAY then inc(ReadyCells);
  lblReady.Caption:= inttostr(ReadyCells);

  //начинаем решение задачи с нулевого уровня:
  Level:= 0; lblLevel.Caption:= inttostr(Level);

  //ряды, в которых есть нулевые группы, можно сразу закрасить белым:
  TestingZero;

  //ряды, в которых есть число, равное размеру поля, можно сразу
  //закрасить чёрным:
  //if TestingFullLine=FALSE then begin SetStatus('ОЖИДАНИЕ'); exit end;

  //если (сумма клеток в группах) + (сумма клеток в группах-1) =
  //длине ряда, то этот ряд разгадан:
  if TestingFullSum=FALSE then begin SetStatus('ОЖИДАНИЕ'); exit end;

  //если единственная группа в ряду длиннее половины ряда,
  //то часть ряда можно закрасить чёрным:
  if TestingLongLine=FALSE then begin SetStatus('ОЖИДАНИЕ'); exit end;

again:
  NoVar:= FALSE;
  //пока в TestingBrain будет закрашена хотя бы одна клетка,
  //продолжаем решение задачи:
  while TestingBrain do begin
    application.ProcessMessages;
    if flgExit=true then begin
```

```
      flgExit:=false;
      SetStatus('ОЖИДАНИЕ'); exit
    end;
    if IsReady then //- все клетки закрашены!
    begin
      //проверить, все ли ряды решены
      //если не все - продолжить:
      for i:= 0 to POLE_WIDTH-1 do
        if masColsNum[i,-1].StatusGroup<> stGreen then goto again;
        break; // - все ряды решены
    end;
  end;

  if IsReady and (NoVar=FALSE) then //- готово!
  begin
    inc(numVar);
    WavPobeda;
    s:='Задача решена!';
    if numVar> 1 then s:= s+ #10#13+ ' Вариант - ' + inttostr(numVar);
    //сложность решения задачи:
    s:= s+ #10#13+ ' Сложность = ' + inttostr(maxLevel+1);
    application.MessageBox(PChar(s),NAME_PROG,MB_OK );
    if Level = 0 then begin
      SetStatus('ОЖИДАНИЕ'); exit
    end;
    //могут быть варианты:
    if application.MessageBox('Ищем варианты?',NAME_PROG, MB_YESNO)= ID_NO
    then begin
      SetStatus('ОЖИДАНИЕ'); exit
    end
    else NoVar:= TRUE
  end;
  //не удалось решить задачу на этом уровне
  if NoVar= TRUE then begin
    dec(Level);
    if Level< 0 then begin
      if numVar= 0 then
        s:='Задача решений не имеет!'
      else
        s:='Найдены все варианты решения задачи - '+ inttostr(numVar);
      application.MessageBox(PChar(s),NAME_PROG,MB_OK );
      SetStatus('ОЖИДАНИЕ'); exit
    end;
    //уровень сложности:
    lblLevel.Caption:= inttostr(Level);
    //загружаем данные предыдущего уровня:
    LoadDataLevel(Level);
    lblReady.Caption:= inttostr(ReadyCells);
    goto again
  end;

  //переходим на следующий уровень -
  //запомнить данные текущего уровня и
  //заменить первую серую клетку на чёрную:
  SaveDataLevel(Level);

  inc(Level);
  if Level > MAX_LEVEL then begin
    s:='Слишком сложная задача!';
    application.MessageBox(PChar(s),NAME_PROG,MB_OK );
    SetStatus('ОЖИДАНИЕ');
    exit
  end;
```

```
  lblLevel.Caption:= inttostr(Level);
  if Level > maxLevel then maxLevel:= Level;
  goto again
end; // sbtStartClick
```

Чтобы защитить программу от шаловливых ручек (от них и самолёты падают!), следует запретить нажимать кнопки *sbtMove* и *sbtDraw* во время решения задачи:


```
//НЕ ВКЛЮЧАТЬ РЕЖИМ ПЕРЕМЕЩЕНИЯ ПРИ РЕШЕНИИ ЗАДАЧИ
procedure TForm1.sbtMoveClick(Sender: TObject);
begin
  if status='ПОИСК' then sbtMove.Down:= FALSE;
end;

//ПЕРЕКЛЮЧИТЬ РЕЖИМ: ВВОД ЗАДАЧИ - РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.sbtDrawClick(Sender: TObject);
begin
  if status='ПОИСК' then sbtDraw.Down:= FALSE;
end;
```

Мастерская Самоделкина, или Наши весёлые картинки

Берите в руки карандаш…

Кавээновская песенка

Когда вы перерешаете сотню-другую японских кроссвордов и у вас возникнет стойкое отвращение к этим головоломкам, попробуйте продлить очарованье и подобно киношному Ивану Васильевичу смените «профессию»: вместо того, чтобы решать чужие задачи, *попридумывайте* свои.

В программе есть всё необходимое для осуществления ваших творческих порывов, всё, за исключением, пожалуй, главного – рисовать картинки придётся вам самим, тут вам никакой компьютер не поможет. Зато с технической стороны нет никаких препятствий. Живописать можно аналогично тому, как вы вручную решали задачи. Только теперь над вами не довлеет «групповщина» чужих клеток, и вы целиком можете предаться воплощению своих самых смелых фантазий.

Нажмите кнопку *sbtNewFig* и установите размеры поля побольше (их всегда можно безнаказанно изменить в процессе работы, так что не впадайте в истерику, если монументальность вашего полотна затмит разум), очистите его от всякой дряни (рекламное словечко, удачно ввернул!)

кнопкой *sbtWhiteGrid* . Включите режим рисования кнопкой *sbtDraw* и приступайте к своим художествам. Чёрные точки и линии наносятся на «холст» *левой* кнопкой мышки, а роль ластика играет *правая* кнопка того же грызуна (ежели у вас толчковая – правая и вы изменили настройки, то тогда, сами понимаете, всё наоборот).

Если вы неудачно разместили картинку на поле, то вдавите поглубже кнопку *sbtMove* и, нажав *левую* кнопку мыши на игровом поле, двигайте её в нужном направлении.

Удовлетворившись содеянным, подгоните размеры поля под картинку так, чтобы не осталось пустых рядов, и сохраните свою работу на диске. Для этого вы можете оцифровать рисунок кнопкой *sbtNumbers* и записать только числа (условие задачи) кнопкой *sbtSaveFig* . Но есть и альтернативный вариант – сохранить именно *картинку*. Если вы были внимательны, то помните, что кнопка *sbtSavePic* до сих пор ничем себя не проявила. А всё потому, что она ждала своего часа, который, наконец, пробил.

По тем же причинам, что и раньше, мы не станем изобретать собственного формата для сохранения готового рисунка, а воспользуемся форматом *JCW* из программы Гантверга *Japan2*, тогда ваши задачи можно будет решать и в ней. Но имейте в виду, что файл *JCW* содержит не условие задачи, а только ответ (копию картинки с игрового поля) – для проверки решения (такие файлы используются при ручном разгадывании). Наша же программа при загрузке файлов *JCW* одновременно формирует и условие задачи, которое всегда можно записать в формате *JCP*, то есть из рисунка сделать головоломку. Тем, кто интересуется, как всё это шевелится, советую посмотреть процедуру *LoadJcwFile*, которая находится в обработчике нажатия на кнопку *sbtLoadFig* - *procedure TForm1.sbtLoadFigClick(Sender: TObject);*.

После всех катавасий и круговертей вам не составит особого труда разобраться в технологических процесах, происходящих при поглаживании кнопки *sbtSavePic*:

```
//ЗАПИСАТЬ КАРТИНКУ
procedure TForm1.sbtSavePicClick(Sender: TObject);
var
  F: file of byte;
  fn,s: string;
  i,j: integer;
  b: byte;
begin
  if status='ПОИСК' then exit;
```

```
  //расширение файлов рисунков:
  savedialog1.DefaultExt:='jcw';
  savedialog1.Filter:='Japan puzzle (*.jcw)|*.JCW';
  //записываем в каталог 'FIGURE':
  s:=extractfilepath(application.exename)+'FIGURE\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите рисунок на диск';
  savedialog1.filename:= NameFig;
  if not savedialog1.Execute then exit;
  //имя конечного файла:
  fn:= savedialog1.filename;
  //изменить его, если при записи было выбрано другое имя:
  fn:=ChangeFileExt(fn, '.jcw');
  NameFig:=fn;
  assignfile(f,fn);
  rewrite(f);
  //записать фигуру -
  //высота и ширина фигуры:
  write (f, POLE_WIDTH);
  write (f, POLE_HEIGHT);
  for i:= 0 to POLE_WIDTH-1 do
    for j:= 0 to POLE_HEIGHT-1 do
    begin
      if masPole[i,j]= NUM_BLACK then b:= 2 else b:= 0;
      write (f, b);
    end;
  //закрыть файл:
  closefile(f);
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  form1.Refresh ;
  messagebeep(0)
end; // sbtSavePicClick
```


Чтобы полнее использовать возможности программы по конвертированию рисунков в задачи, в ней предусмотрена загрузка растровых картинок *BMP* и значков *ICO* в игровое поле. Одновременно изображение появится и на компоненте *Image1* (*frmMemo*), чтобы оригинал всегда был у вас перед глазами.

Предупреждения.

1. Избегайте картинок больших размеров, всё равно вы не сможете их загрузить. В данной версии программы максимальные размеры растров не должны превышать 70 х 50 пикселей (вы, конечно, можете и «раздвинуть границы», если пожелаете).

2. Если картинки не монохромные, то все цвета за исключением белого и чёрного будут заменены *серым*. Вы должны перекрасить их в чёрный и белый цвет так, что-

бы получился рисунок. Поручить это компьютеру нельзя, он скорее маляр, чем художник, так что результаты почти наверняка окажутся отвратительными.

А теперь полюбуйтесь, как в программе совершается таинство преобразования глупого растра в полную глубокого смысла картинку на поле:

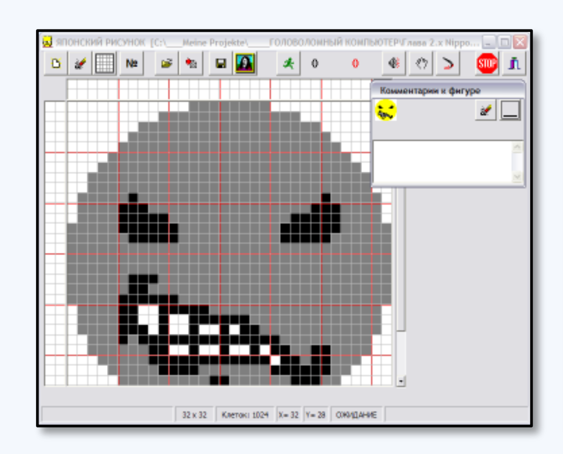

```
//ЗАГРУЗИТЬ РАСТРОВУЮ КАРТИНКУ ИЛИ ЗНАЧОК
procedure TForm1.sbtOpenPictureClick(Sender: TObject);
var
  w, h: integer;
  i, j: integer;
  bmp: TBITMAP;
begin
  frmMemo.Image1.Visible:= FALSE;
  OpenPictureDialog1.DefaultExt:= 'BMP';
  OpenPictureDialog1.InitialDir:= extract-
filepath(application.exename);
  OpenPictureDialog1.Title:='Загрузите картинку';
  OpenPictureDialog1.Filter:= GraphicFil-
ter(TBitmap)+'|'+GraphicFilter(TIcon);
  if not OpenPictureDialog1.Execute then exit;
  //имя файла:
  NameFig:= OpenPictureDialog1.FileName;
  //загрузить картинку:
  frmMemo.Image1.Picture.LoadFromFile(NameFig);
  if OpenPictureDialog1.FilterIndex= 2 then begin //- значок
    bmp:= TBitmap.Create;
    bmp.Width:= frmMemo.Image1.Picture.icon.Width;
    bmp.Height:= frmMemo.Image1.Picture.icon.Height;
    bmp.Canvas.Draw(0,0,frmMemo.Image1.Picture.icon);
    frmMemo.Image1.Picture.Bitmap.Assign(bmp);
    bmp.Free;
  end;
  w:= frmMemo.Image1.Picture.Width;
  h:= frmMemo.Image1.Picture.Height;
  //проверить размеры картинки:
  if (h > MAX_POLE_HEIGHT) or (w > MAX_POLE_WIDTH)then begin
    application.MessageBox('Слишком большая картинка!',NAME_PROG,
MB_OK);
    exit
  end;
  frmMemo.Image1.Visible:= TRUE;
  POLE_WIDTH:= w; POLE_HEIGHT:= h;
  TOP_POLE_HEIGHT:= 2; LEFT_POLE_WIDTH:= 2;
  Prepare(POLE_WIDTH,POLE_HEIGHT, TOP_POLE_HEIGHT,
LEFT_POLE_WIDTH);
  //очистить цифровые поля:
  Clear_masColsNum;
  Clear_masRowsNum;
  for j:= 0 to POLE_HEIGHT-1 do
```

```
    for i:= 0 to POLE_WIDTH - 1 do begin
      case frmMemo.Image1.Picture.Bitmap.Canvas.Pixels[i,j] of
        clBlack: masPole[i,j]:= NUM_BLACK;
        clWhite: masPole[i,j]:= NUM_WHITE;
        else masPole[i,j]:= NUM_GRAY;
      end;
  end;
  //вывести в заголовке формы имя загруженного файла:
  form1.caption:= NAME_PROG + '  [' + NameFig + ']';
  dgPole.Invalidate
end; // sbtOpenPictureClick
```


Раз уж вы можете загружать в программу растры и значки, то лучше создавать картинки-задания в каком-нибудь растровом графическом редакторе или в редакторе иконок (их великое множество, выбирайте на свой вкус!). Они значительно удобнее и мощнее, чем наша скромная в этом отношении программа (всё-таки не царское это дело, рисование; вот и Остап Ибрагимович рисовал как курица лапой, а ведь был милейшим человеком). А подготовив рисунок, просто загрузите его кнопкой *sbtOpenPicture*, а затем сохраните как картинку *JCW* или как задачу *JCP* для последующего применения.

Вы легко найдёте массу небольших картинок на своём диске или в Интернете, которые вполне можно, приложив руки и сердце, переделать в японские кроссворды. На первых порах это невинное занятие поможет вам приобрести необходимый для создания будущих шедевров опыт.

А вот наглядный пример хорошего отношения к людям. Слева показана форма, в которую загружен значок с изображением колобка. Справа – он же, но раскрашенный и оцифрованный (Рис. 3.39).



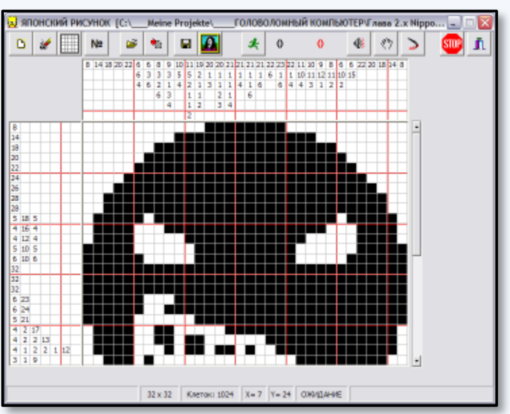


Рис. 3.39. Оцифровали!

Лёгким движением руки значок превратился - в японский кроссворд (знали бы японцы, что мы сделаем с их творением!). Тот, кто способен на боль-

шее, чем примитивный «плагиат», может изувечить оригинал до полной неузнаваемости. А те и другие сообща имеют полное право наклонировать теперь бешеное количество задач, не имеющих практически никаких достоинств.

«Шлите апельсины бочками», или «Ждите ответа!»

Ищите – и обрящете!

Библейское

Не очень-то увлекательно составлять задачи для самого себя, лучше позабавить ими кого-нибудь другого. Благо, существует немало изданий (смотри выше), отдающих должное японским кроссвордам. Вы вполне можете послать свои работы в редакцию и, если вам удалось в грубых квадратах выразить свои изящные замыслы, у вас есть последняя надежда увидеть свои кроссворды на страницах какой-нибудь газеты. Это поднимет ваш боевой дух и толкнёт вас на новые опрометчивые поступки в области сочинения головоломок.

Если же вас утомит сам процесс разработки задания, то просто распечатайте готовый рисунок на принтере, скопировав его предварительно в любой графический редактор. Этого вполне достаточно, чтобы послать задание в газету. А можете и красиво оформить его. Например, чудесно для этого подходит векторный графический редактор *CorelDraw*.

Засим позвольте откланяться – наш художественный салон закрывается.

«А напоследок я скажу…»

На этом можно и закончить изучение программы для составления и решения японских кроссвордов – теперь вы знаете о ней всё. Осталось - на всякий случай – привести начало программы, где объявляются константы, переменные, типы, процедуры и функции, так как это делалось на протяжении многих страниц, и вы могли что-нибудь упустить в пылу борьбы за урожай(ность).

```
const
  NAME_PROG = 'ЯПОНСКИЙ РИСУНОК';
  MAX_POLE_WIDTH = 70;    //макс. ширина поля в клетках
  MAX_POLE_HEIGHT = 50;   //макс. высота поля в клетках
  NUM_WHITE = 2     ;     //пустая клетка в masPole
  NUM_BLACK = 1;          //чёрная клетка в masPole
```

```
  NUM_GRAY= 0;           //клетка серая в masPole (фон)
  COLOR_WHITE: TColor= $FFFFFF;  //белая клетка на поле
  COLOR_BLACK: TColor= $0;       //чёрная клетка на поле
  COLOR_GRAY: TColor= $7F7F7F;   //серая клетка на поле
  MAX_CELLNUM = 14;          //макс. ширина/высота числовых полей
10
  MAX_ALL_CELLS_HEIGHT = 30;//макс. высота игрового и числового
полей
                            //(видимая часть)
  MAX_ALL_CELLS_WIDTH = 46; //макс. ширина игрового и числового
полей
  //курсоры:
  crHand :integer= 4;
  crMove :integer= 5;
  crKistj :integer= 7;
  MAX_LEVEL= 20;       //макс. сложность задачи (число предположе-
ний)

//данные игрового поля
type TPole = array[0..MAX_POLE_WIDTH-1, 0..MAX_POLE_HEIGHT-1]of
Integer;
//статус групп в числовых полях:
//stWhite - группа не решена,
//stYellow - группа решена,
//stGreen - ряд решён
type TStatusGroup=(stWhite, stYellow, stGreen);
//данные в каждой клетке числовых полей:
type TCell = Record
  sNum: String; //строковое представление числа в клетке
   Num: integer;//число в клетке
   StatusGroup: TStatusGroup; //статус группы
end;
//положение числового поля: слева, сверху
type TPoleLocation= (plLEFT, plTOP);
//данные уровня
type TSaveLevel = Record
  //статус рядов верхнего числового поля:
  TopStatus: array[0..MAX_POLE_WIDTH-1]of TStatusGroup;
  //статус рядов левого числового поля:
  LeftStatus: array[0..MAX_POLE_HEIGHT-1]of TStatusGroup;
  //копия поля:
  masPole: TPole;
  //коорд. "сомнительной" клетки
  XCell: integer;
  YCell: integer;
  //число разгаданных клеток:
  ReadyCells: integer;
end;

var
  //массив, в котором хранится копия поля
  masPole : TPole;
  POLE_WIDTH: integer = 28;     //ширина поля в клетках
```

```
  POLE_HEIGHT: integer = 28;    //высота поля в клетках
  //массив, в котором хранятся верхние числа:
  masColsNum : array[0..MAX_POLE_WIDTH-1, -1..MAX_CELLNUM-1]of 
TCell;
  //массив, в котором хранятся левые числа:
  masRowsNum : array[-1..MAX_CELLNUM-1, 0..MAX_POLE_HEIGHT-1]of 
TCell;
  LEFT_POLE_WIDTH: integer = 2; //ширина левого числового поля в 
клетках
  TOP_POLE_HEIGHT: integer = 2; //высота верхнего числового поля в 
клетках
  status: string='';
  NameFig: string= 'temp';
  //координаты клетки с курсором:
  cellMouse: TPoint;      //игровое поле
  cellMouseTop: TPoint;   //верхнее числовое поле
  cellMouseLeft: TPoint;  //левое числовое поле
  AllCells: Integer=0;    //всего клеток на поле
  ReadyCells: Integer=0;  //разгадано клеток
  flgExit: boolean= False;
  Level: Integer=0;       //уровень сложности задачи (число предпо-
ложений)
  //массив, в котором хранятся данные предыдущих уровней:
  SaveLevel: array[0..MAX_LEVEL] of TSaveLevel;

type
  TForm1 = class(TForm)
     . . .
  private
    { Private declarations }
    procedure Clear_masColsNum;
    procedure Clear_masRowsNum;
    procedure InvalidateGrids;
    procedure MoveNums;
    procedure WavError;
    procedure WavReadyLine;
    procedure WavPobeda;
    function GetNumGroup(Location: TPoleLocation; nLine: integer): 
integer;
    function GetLenGroup(Location: TPoleLocation; nLine, nGroup: 
integer)
             : integer;
    procedure SetStatus (s: string);
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

uses MemoUnit, NewFigUnit;
```

```
{$R *.DFM}
. . .
```



Исходный код программы находится в папке **NIPPON**.

Генеалогическое древо, или Японская родня

Гора родила мышь.

Нашему забору двоюродный плетень.

Из мудрых мыслей и афоризмов

Как и всякая другая известная головоломка, японский кроссворд имеет бедных родственников, к счастью, не столь многочисленных. Пока все дома, давайте познакомимся с несколькими из них.

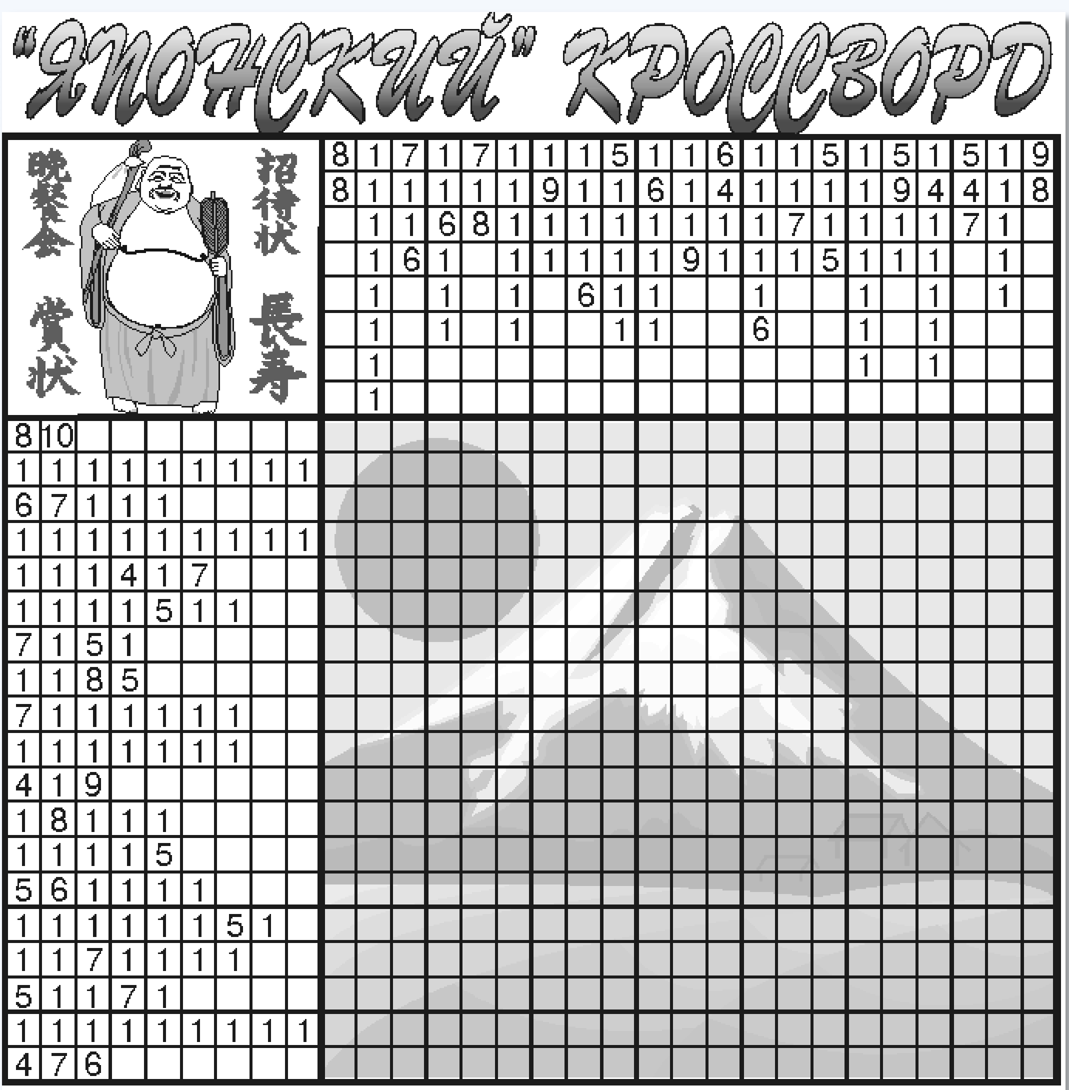
В последние годы японский кроссворд мутировал, породив химеру – *цветной* рисунок, в котором клетки могут быть не только чёрными и белыми, но и вообще любых мыслимых (и немыслимых) цветов. Второе отличие от классического рисунка состоит в том, что между группами цветных клеток прослойки в виде белых клеток не обязательны.

Решается подобное чудо природы практически так же, как и обычная задача, хотя забава куда как веселее, ведь для раскрашивания клеток придётся держать в руках целый пучок карандашей!

Так как цветные рисунки не имеют отношения к нашим досугам, то писать программу для их решения мы не станем. Если же вам неймётся, не терпится и не хочется ждать милостей от мерзкой природы – милости прошу, займитесь-ка этим сами. Чтобы не изобретать велосипед (или того хуже – самокат), попробуйте «уговорить» нашу программу на «цветное» решение. Но будьте готовы к большой работе!

Вторая, более светлая мысль авторов привела к *гибридизации*, то есть к скрещиванию японского кроссворда с обычным (который, если вы не знаете, является для нас важнейшим из всех искусств). Метис получился презанятным, но бесплодным. Посему, не выдержав напора эволюции, сгинул,

как и все прочие динозавры. Так как автор этих строк и междустрочий тоже принимал непосредственное участие в выведении несуразных монстров, то – исключительно в кунсткамерных целях – решился на демонстрацию своей напечатанной в одной из московских газет работы (Рис. 3.40).



8	1	7	1	7	1	1	1	5	1	1	6	1	1	5	1	5	1	5	1	9
8	1	1	1	1	1	9	1	1	6	1	4	1	1	1	1	9	4	4	1	8
	1	1	6	8	1	1	1	1	1	1	1	1	7	1	1	1	1	7	1	
	1	6	1		1	1	1	1	1	9	1	1	1	5	1	1	1		1	
	1		1		1		6	1	1			1			1		1		1	
	1		1		1			1	1			6			1		1			
	1														1		1			
	1																			

8	10							
1	1	1	1	1	1	1	1	1
6	7	1	1	1				
1	1	1	1	1	1	1	1	1
1	1	1	4	1	7			
1	1	1	1	5	1	1		
7	1	5	1					
1	1	8	5					
7	1	1	1	1	1	1		
1	1	1	1	1	1	1		
4	1	9						
1	8	1	1	1				
1	1	1	1	5				
5	6	1	1	1	1			
1	1	1	1	1	1	5	1	
1	1	7	1	1	1	1		
5	1	1	7	1				
1	1	1	1	1	1	1	1	1
4	7	6						

По правилам головоломки "Японский рисунок" восстановите сетку кроссворда, а затем заполните её словами, определения которых даны в произвольном порядке.

4-буквенные слова: 1."Пуля - дура, … - молодец" (Суворовское). 2. Необъяснимое явление. 3. Английский парный танец. 4. "Воздухоплаватель", оторвавшийся от коллектива. 5. Резкое снижение самолёта. 6. Жёлоб.

5-буквенные слова: 7. Съедобный гриб "хитрого" цвета. 8. Начало реки. 9. Схватка за урожай. 10. Руководитель факультета. 11. Аркан. 12. Тара для переноски котов. 13. Осёл и катапульта. 14. Вересковый кустарник. 15. Резвый, игривый ребёнок. 16. Землечерпалка. 17. Обезжиренное молоко. 18. Инструмент для прополки.

6-буквенные слова: 19. Заболевание суставов. 20. Детёныш яка. 21. Гигант на глиняных ногах. 22. Радуга как набор цветов. 23. Бегун-дальнобойщик. 24. Поэма Лермонтова о предыдущем пункте. 25. Сигаретный "бычок". 26. Угнетённое состояние. 27. Скульптура.

7-буквенные слова: 28. Вирусная болезнь глаз. 29. Участник корриды с копьём. 30. Наука для проверки гармонии. 31. Осколочная … 32. Неявка на работу. 33. Невезение, провал. 34. Маленькая … до старости щенок (посл.). 35. Охотничья плеть. 36. Приучение лошади к езде. 37. Писатель, который не попадает в рифму. 38. "Поцелуй же меня, кума-…" (песенное).

8-буквенные слова: 39. Праздник в Рио. 40. Один из тех, что съели Кука. 41. На все ноги мастер. 42. Комнатный беспорядок. 43. Чиполлино как ботанический объект. 44. Прослойка между бедняком и кулаком. 45. Каждый из неуловимых.

9-буквенные слова: 46. Визави сталагмита. 47. Жительница чёрного континента. 48. Невосприимчивость к заразе. 49. Жительница одной из стран Юго-Восточной Азии. 50. Периодическое издание.

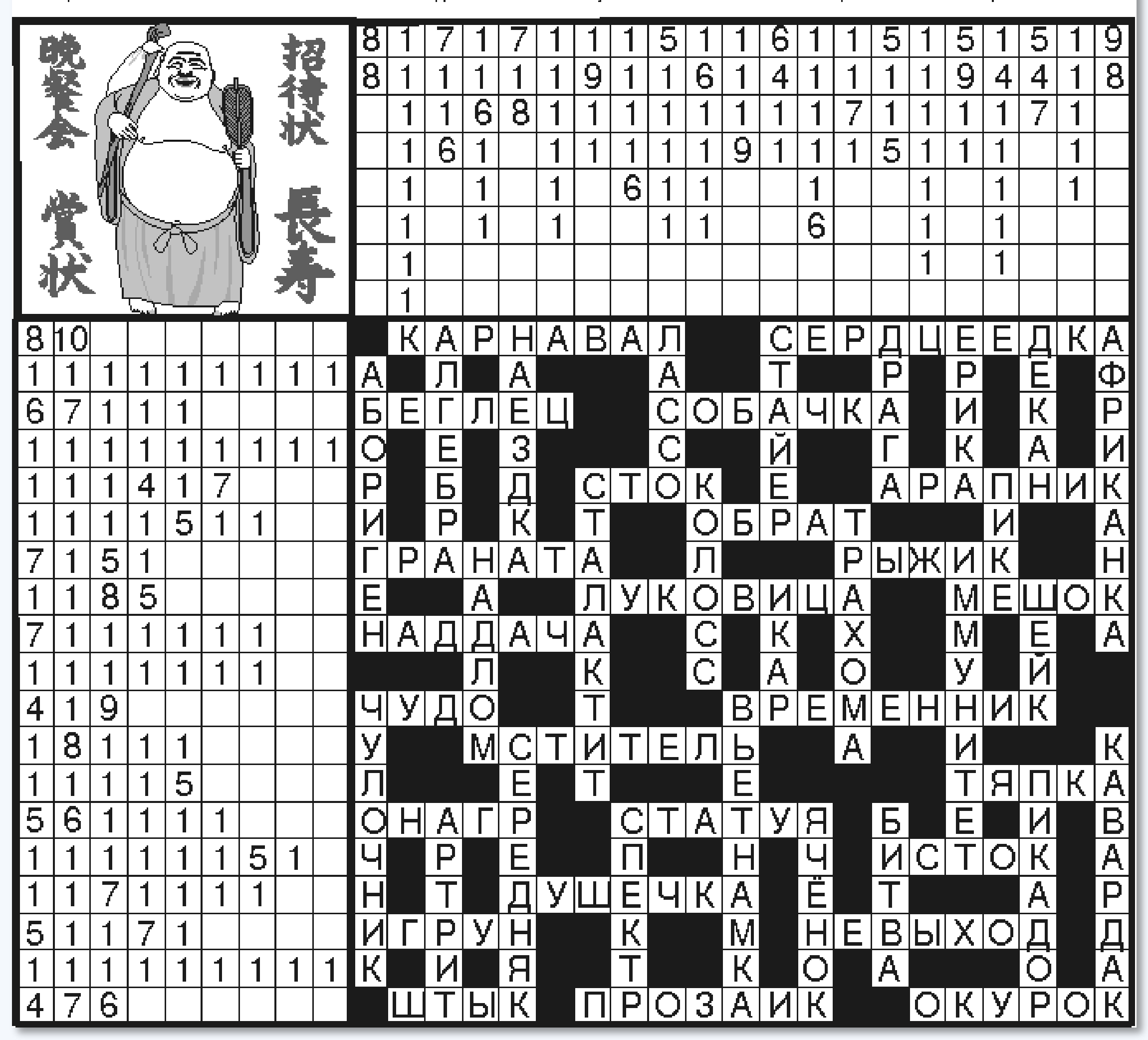


Рис. 3.40. А что, неплохо получилось!



Наша программа напичкана всем, что нужно для плодотворной работы. Но вы можете на досуге подумать вот над чем. Почти во всех процедурах и функциях программы одни и те же действия проводились отдельно для верхнего и левого числовых полей. Сделано это исключительно для простоты понимания, но теперь это вам уже ни к чему и вы спокойно можете переделать программу так, чтобы не писать дважды один и тот же код. Пример этого вы найдёте в самой программе. Например, функция *TestingOneLine* обрабатывает и вертикальные, и горизонтальные ряды. О других усовершенствованиях программы уже говорилось раньше. Кроме того, вы можете проявить разумную инициативу и даже отчаянную смелость на этом поприще.

Факультатив 4. Игра-головоломка *Logos*

Logos - это прекрасная, оригинальная компьютерная головоломка! Она требует от игрока минимальных знаний, поэтому в неё могут играть даже маленькие дети. И в то же время игра *Logos* прекрасно развивает логическое и комбинаторное мышление.

Правила игры

очень просты и немногочисленны:

- На клетки прямоугольного поля поочерёдно выставляются фишки, которые можно представить в виде игральных кубиков. На верхней, видимой грани «свежего» кубика находится *одно* очко. На передней – *два*, на нижней – *три* и, наконец, на задней – *четыре*. Боковые грани в игре не участвуют.

- Если вновь выставленная фишка соприкасается своими гранями с уже имеющимися на поле, то последние поворачиваются так, чтобы на их верхней грани оказалось на единицу большее число точек (очков). После четырёх очков, естественно, на верхнюю грань снова выйдет одно очко.

Например, на доске (игровом поле) стоит фишка (Рис. 4.1).

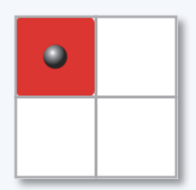


Рис. 4.1. Занимаем исходное положение!

- Если следующие фишки выставляются так, что они не имеют *общих* граней с фишками на поле, то эти фишки не изменяют своих значений (Рис. 4.2).

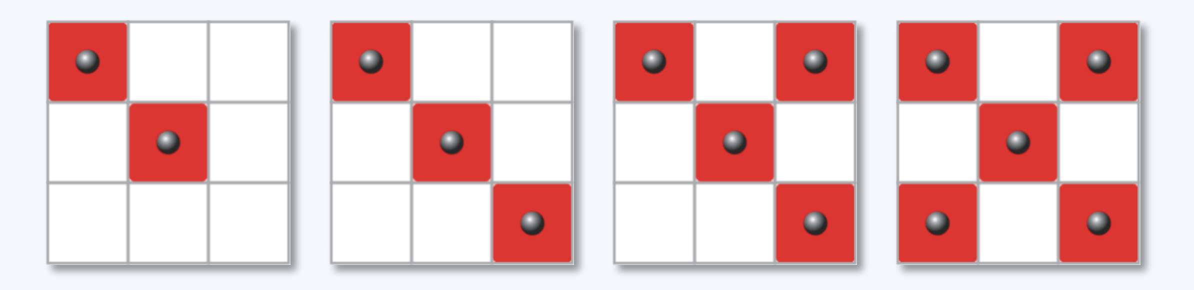


Рис. 4.2. Не тревожьте соседей!

- Если же новая фишка *соприкасается* с ранее выставленными фишками, то они поворачиваются на следующую грань (Рис. 4.3).

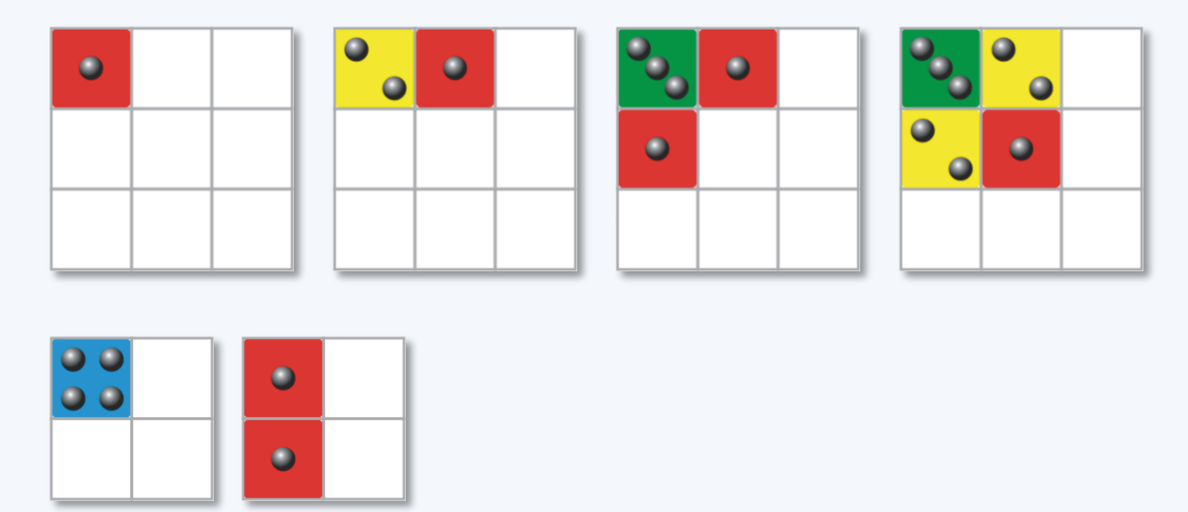


Рис. 4.3. Поворачиваемся с боку на бок!

Цель игры: выставляя на поле фишки с соблюдением правил, построить заданную фигуру.

Далее мы подробно разберём алгоритм игры, позволяющий достаточно легко решить *любую* задачу, а пока попробуйте самостоятельно справиться с несколькими самыми простыми заданиями.

Составьте следующие фигуры, последовательно выставляя на первоначально пустое поле фишки.

Задача 1. (Рис. 4.4).

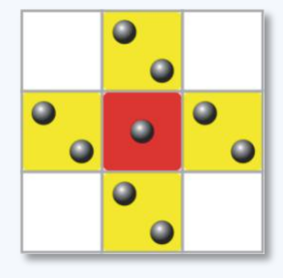


Рис. 4.4. Вот и первая задача!

Задача 2. (Рис. 4.5).

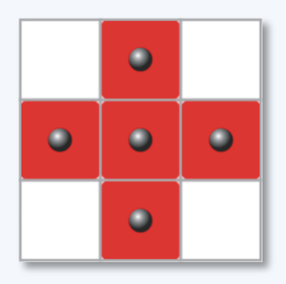


Рис. 4.5. Задача не приходит одна!

Задача 3. (Рис. 4.6).

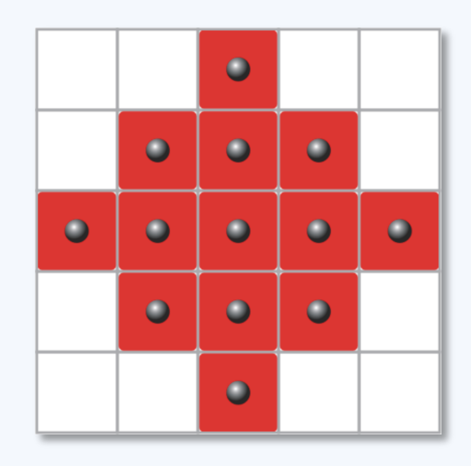


Рис. 4.6. Задачи ходят по-трое!



Программируем от достигнутого

Так как правила игры *Logos* не очень сильно отличаются от правил игры *ShortGame*, то мы возьмем за основу интерфейс этой игры, только исправим и дополним его новыми элементами. Поэтому начните работу над новым проектом с того, что загрузите в *Turbo Delphi*, а затем сохраните в новой папке все файлы игры *ShortGame*.

Правила игры *Logos* никак не ограничивают размеры игрового поля, но опыт показывает, что 13 клеток по горизонтали и столько же по вертикали вполне достаточно для любых разумных заданий. Это немного больше, чем в игре *ShortGame*, поэтому и форму, и поля придётся увеличить, а размеры ячеек сеток - уменьшить. Кроме того, форму необходимо «адаптировать» к новой игре – заменить *название* и *значок* в заголовке:

```
BorderIcons = [biSystemMenu, biMinimize]
```

```
  BorderStyle = bsSingle
  Caption = '  Logos'
  ClientHeight = 515
  ClientWidth = 726
  Color = clSkyBlue
  PopupMenu = PopupMenu1
  Position = poDesktopCenter

  OnCreate = FormCreate
  OnMouseMove = FormMouseMove
  OnShow = FormShow
```


Совсем немного придётся поработать и над обеими *сетками*.

Сетка dgPole:

```
    Left = 4
    Top = 44
    Width = 465
    Height = 465
    Cursor = crHandPoint
    ColCount = 14
    RowCount = 14 (не забывайте об оцифровке!)
    DefaultColWidth = 32
    DefaultRowHeight = 32
    DefaultDrawing = False
    FixedCols = 0
    FixedRows = 0
    Options = [goVertLine, goHorzLine]
    ScrollBars = ssNone

    OnDrawCell = dgPoleDrawCell
    OnMouseDown = dgPoleMouseDown
    OnMouseMove = FormMouseMove
```

Сетка dgPole2:

```
    Left = 484
    Top = 44
    Width = 241
    Height = 241
    Cursor = crHandPoint
    ColCount = 14
    RowCount = 14
    DefaultColWidth = 16
    DefaultRowHeight = 16 (размер клеток на правом поле устанавли-
ваем в два раза меньше, чем на игровом)
    DefaultDrawing = False
    Options = [goVertLine, goHorzLine]
    ScrollBars = ssNone

    OnDrawCell = dgPole2DrawCell
```

```
    OnMouseDown = dgPole2MouseDown
    OnMouseMove = FormMouseMove
```

Соответствующим образом изменяем и все глобальные *константы*:

```
const
  //макс. размеры поля:
  MAX_POLE_WIDTH= 14;
  MAX_POLE_HEIGHT= 14;
  NAME_PROG='Logos';
  letter: string= 'ABCDEFGHIJKLM';
```


Практически без изменений останутся объявления *типов*:

```
type
  TPole=   array[0..MAX_POLE_WIDTH-1,   0..MAX_POLE_HEIGHT-1]   of
Shortint;
  TMemory= Record
    Hod: integer;
    //позиция на полях:
    PosL, PosR: TPole;
  end;
  //состояние программы:
  TGameState= (gsWait, gsSolution);
  //ходы:
  TMoves= Record
    //колонка:
    Col: integer;
    //строка:
    Row: integer;
    //позиция на полях:
    PosL, PosR: TPole;
  end;
```

Для большинства глобальных *переменных* игры *ShortGame* найдётся работа и в нашей программы, а остальные мы будем добавлять по мере необходимости:

```
var
  //поля:
  masPole, masPole2: TPole;
  Memory: TMemory;
  GameState: TGameState= gsWait;
  //запись ходов:
  Moves: array[0..169] of TMoves;
  //номер хода:
  Hod: integer= 0;
  Hod_max: integer= 0;
  step: integer;
  rep: integer= 0;
```

Поскольку размеры формы и полей в этой программе *постоянны*, то подготовка к новой игре намного облегчается. Достаточно обнулить ходы и очистить оба поля:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //подготовить новую игру:
  NewPlay;
end; //FormCreate

//ПОДГОТОВКА К НОВОЙ ИГРЕ
procedure TForm1.NewPlay;
var
 i, j: integer;
begin
  //обнулить число ходов:
  Hod:= 0;   Hod_max:= 0;
  lblHod.Caption:=  'Ход - 0';

  //задача не имеет названия:
  form1.caption:= NAME_PROG + '   []';

  //очистить левое поле:
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do
      masPole[i,j]:= 0;
  dgPole.Invalidate;

  //очистить правое поле:
  masPole2:= masPole;
  dgPole2.Invalidate;
  //запомнить начальную позицию:
  MemoryIn;
end; // NewPlay
```


В массивах *masPole* и *masPole2 нулём* мы будем обозначать *пустую* клетку поля, а числами *1, 2, 3, 4* – количество очков на верхней грани фишки. Всё настолько понятно, что даже не нужно вводить констант для содержимого клеток полей.

Изменившиеся размеры ячеек и игрового поля заставляют нас нарисовать новые *картинки* для оцифровки поля (первая картинка в программе не используется) (Рис. 4.7)

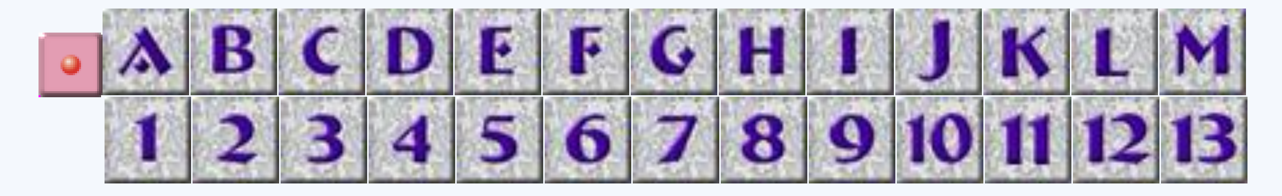


Рис. 4.7. Оцифровка поля

Как и раньше, все эти картинки нужно загрузить в компонент *imlNota* типа *TImageList*, предварительно установив их размер:

```
Height = 32
Width = 32
```

Рисунки фишек и клеток поля (Рис. 4.8) мы разместим в компонентах *TImage* на форме (если вам больше нравятся списки *TImageList*, то вы легко можете подправить код программы).

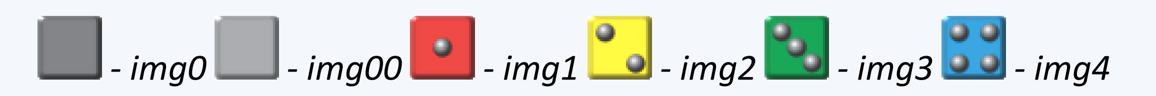


Рис. 4.8. Фишки-малышки

Для удобства решения (и «красивости») грани фишек окрашены в *разные* цвета, а клетки поля имеют разную интенсивность окраски. Те клетки игрового поля, которые соответствуют клеткам *правого* поля с фишками, мы будем окрашивать *светлее*, чтобы выделить «игровые» клетки, в которые следует ходить.

Верхний левый угол поля мы по-прежнему используем для анимации, только сделаем её более «изощрённой» - там будет пульсировать кубик с четырьмя точками. Три картинки (Рис. 4.9) для анимации мы загрузим в компонент *TImageList imlAnima* (не забудьте установить размер картинок 32 на 32 пикселя!).

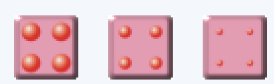


Рис. 4.9. Четырёхточечный «пульсар»

Номер текущей (видимой) картинки хранится в глобальной *переменной*:

```
flgFlash: integer= 0;
```

Она последовательно принимает значение 0, 1, 2, 0, 1, 2, ... , в соответствии с которым и выводится в ячейку картинка из списка. Частота пульсации определяется свойством *Interval* таймера. Неплохие результаты получаются, если установить *Interval = 100*.

```
//ПУЛЬСИРОВАНИЕ УГЛОВОГО КВАДРАТИКА
procedure TForm1.Timer2Timer(Sender: TObject);
var
  rect: TRect;
  n: integer;
begin
```

```
  //угловой квадратик поля:
  rect:= dgPole.CellRect(0,0);
  //номер картинки:
  case flgFlash of
    0,1: inc(flgFlash);
    2: inc(flgFlash);
    3: flgFlash:= 0;
  end;
  if flgFlash= 3 then n:= 0 else n:= flgFlash;
  imlAnima.Draw(dgPole.Canvas, rect.left, rect.top, n);
end;  //Timer2Timer
```


Других премудростей в процедуре рисования ячейки игрового поля нет:

```
//ОТРИСОВАТЬ ЯЧЕЙКУ ИГРОВОГО ПОЛЯ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var
  n, m: integer;
  r, dr: TRECT;
begin
  //какая фишка в ячейке:
  n:= masPole[ACol, ARow];
  //размеры картинок:
  r:= Bounds(0, 0, 32, 32);
  dr:= Bounds(Rect.Left, Rect.Top, 32, 32);
  m:= 100 * ARow + ACol;
  case m of
    0..13: //- верхняя строка
      imlNota.Draw(dgPole.Canvas, dr.left, dr.top, m);
    100, 200, 300, 400, 500, 600, 700, 800,
900,1000,1100,1200,1300: //- первая колонка
      imlNota.Draw(dgPole.Canvas, dr.left, dr.top, m div 100+13);
    else begin
      case n of
        0:    //- пустая клетка
          if masPole2[ACol, ARow]<> 0 then
            dgPole.canvas.CopyRect(dR, img00.Canvas, R)
          else
            dgPole.canvas.CopyRect(dR, img0.Canvas, R);
        1:   //- фишка 1
          dgPole.canvas.CopyRect(dR, img1.Canvas, R);
        2:   //- фишка 2
          dgPole.canvas.CopyRect(dR, img2.Canvas, R);
        3:   //- фишка 3
          dgPole.canvas.CopyRect(dR, img3.Canvas, R);
        4:   //- фишка 4
          dgPole.canvas.CopyRect(dR, img4.Canvas, R);
      end
    end
  end;
```

```
end;  //dgPoleDrawCell
```

Из-за различия в размерах ячеек *правого* поля и рисунков с оцифровкой и фишками нам придётся картинки «ужимать». С картинками-фишками всё достаточно просто, так как мы можем применить к ним для вывода в ячейки удобный метод канвы *StretchDraw* (поэтому, для сравнения, эти картинки и размещены в компонентах *TImage*, а не в *TImageList*). Но компонент *TImageList* такого метода не имеет и выводит картинки только того размера, который задан его свойствами *Height* и *Width* (то есть 32 на 32 пикселя), а размеры ячеек *правого* поля в два раза меньше (16 на 16 пикселей) – тут нам никак не избежать дополнительного растра *bmp* в памяти компьютера. Сначала мы скопируем картинку из списка его методом *Draw* в растр, а уже оттуда – в нужную ячейку сетки с масштабированием, используя метод канвы *StretchDraw*. Слегка замысловато, но зато не нужно заготавливать два комплекта рисунков разного размера.


```
//ОТРИСОВАТЬ ЯЧЕЙКУ ПОЛЯ-ОБРАЗЦА
procedure TForm1.dgPole2DrawCell(Sender: TObject; ACol, ARow: In-
teger;
  Rect: TRect; State: TGridDrawState);
var
  n, m: integer;
  r, dr: TRECT;
  bmp: TBitmap;
begin
  //фишка в ячейке:
  n:= masPole2[ACol, ARow];
  //размеры картинок:
  r:= Bounds(0, 0, 16, 16);
  dr:= Bounds(Rect.Left, Rect.Top, Rect.Right-
Rect.Left,Rect.Bottom-Rect.Top);

  m:= 100 * ARow + ACol;

  bmp:= TBitmap.Create;
  bmp.Width:= 32;
  bmp.Height:= 32;

  case m of
    0..13: //- верхняя строка
      begin
        imlNota.Draw(bmp.Canvas, 0, 0, m);
        dgPole2.canvas.StretchDraw(dR, bmp);
      end;
    100, 200, 300, 400, 500, 600, 700, 800,
900,1000,1100,1200,1300: //- первая колонка
      begin
        imlNota.Draw(bmp.Canvas, 0, 0, m div 100 + 9);
```

```
        dgPole2.canvas.StretchDraw(dR, bmp);
      end;
    else begin
      case n of
        0:     //- пустая клетка
          dgPole2.canvas.StretchDraw(dR, img0.Picture.Graphic);
        1:    //- фишка 1
          dgPole2.canvas.StretchDraw(dR, img1.Picture.Graphic);
        2:    //- фишка 2
          dgPole2.canvas.StretchDraw(dR, img2.Picture.Graphic);
        3:    //- фишка 3
          dgPole2.canvas.StretchDraw(dR, img3.Picture.Graphic);
        4:    //- фишка 4
          dgPole2.canvas.StretchDraw(dR, img4.Picture.Graphic);
      end
    end
  end;
  bmp.Free;
  dgPole.Invalidate;
end;  //dgPole2DrawCell
```


Запустив программу, мы увидим *пустые* поля, ведь никто не побеспокоился загрузить задание. Что ж, позаботимся о себе сами. Проще всего расставлять фишки, нажимая *левую* кнопку мыши на нужной клетке поля. Сколько раз кликнул, такая фишка и появилась (место фишки с четырьмя очками займёт пустая клетка, затем с одним очком и так далее). На *левом*, игровом поле будет синхронно появляться бледный образ будущей задачи.

```
//ПОСТАВИТЬ ФИШКУ НА ПОЛЕ-ОБРАЗЦЕ
procedure TForm1.dgPole2MouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
begin
  //координаты мыши:
  dgPole2.MouseToCell(x,y,ACol,ARow);
  //нажата левая кнопка мыши - ставим след. фишку:
  if (ssLeft in shift) then begin
    inc(masPole2[ACol,ARow]);
    if masPole2[ACol,ARow]>4 then masPole2[ACol,ARow]:= 0;
    dgPole2.Invalidate;
  end
end;  //dgPole2MouseDown
```

В качестве примера вы можете ввести задачи для самостоятельного решения.

И вот задача ждёт своего решателя. А он (то есть мы) уже тут как тут – нажимаем левую кнопку мыши в нужной клетке - и фишка с одним глазком сейчас же появляется в ней. Со вторым ходом сложнее – нужно изменить все соседние фишки. Эту операцию мы поручим *функции DoMove*, которая вернёт нам поле со всеми изменениями:

```
 //ВЫПОЛНИТЬ ХОД
function TForm1.DoMove(arr: TPole; x, y: integer): TPole;
var i: integer;
begin
  result:= arr;
  //нельзя ходить в занятую клетку:
  if arr[x,y]<> 0 then begin dec(hod);exit; end;
  //изменить соседние фишки:
  for i:= Max(y-1,1) to Min(y+1,dgPole.RowCount-1) do //- верти-
кальный ряд
  begin
    if arr[x,i] <> 0 then inc(arr[x,i]);
    if arr[x,i]>4 then arr[x,i]:= 1;
  end;
  for i:= Max(x-1,1) to Min(x+1,dgPole.ColCount-1) do //- горизон-
тальный ряд
  begin
    if arr[i,y] <> 0 then inc(arr[i,y]);
    if arr[i,y]>4 then arr[i,y]:= 1;
  end;
  //поставить в заданную клетку фишку 1:
  arr[x,y]:= 1;
  result:= arr;
end; //DoMove
```


Эта функция понадобится нам ещё и в другом модуле, поэтому её объявление следует перенести в раздел *public* типа формы:

```
  public
    { Public declarations }
    function DoMove(arr: TPole; x, y: integer): TPole;
    function IsReady: Boolean;
    procedure Ready;
```

Очень удобно иметь перед глазами запись всех сделанных ходов, как это принято в шахматах. У нас уже имеется форма *frmProtokol* для ведения протокола, так что нам осталось усовершенствовать её так, чтобы она всегда была на экране.

```
  Left = 600
  Top = 332
  BorderIcons = []
  BorderStyle = bsSingle
  Caption = '  Протокол'
```

```
  ClientHeight = 215
  ClientWidth = 191
  Font.Name = 'MS Sans Serif'
  FormStyle = fsStayOnTop  (располагаем форму поверх других окон
приложения)
```

Модуль формы протокола нужно объявить в модуле *LogosUnit*, чтобы сделать его доступным в главном модуле.

```
implementation

uses ProtokolUnit, ReadyUnit;
```


Чтобы форма с протоколом не перекрывала элементы управления на главной форме, выводим её в правом нижнем углу:

```
//ПОКАЗАТЬ ФОРМУ
procedure TForm1.FormShow(Sender: TObject);
begin
  open('001.lgs');
  frmProtokol.show;
end; //FormShow

procedure TForm1.FormActivate(Sender: TObject);
begin
  frmProtokol.Left:= form1.Left + lblHod.Left;
  frmProtokol.Top:= form1.Top + lblHod.Top+lblHod.Height+35;
end; //FormActivate
```

Обратите внимание, что при каждом ходе записывается его номер и координаты клетки – в том виде, как это делается в настольных играх (а иначе зачем мы делали оцифровку полей?). И напоследок нужно проверить, а не решена ли задача. Для этого после каждого хода мы вызываем *функцию IsReady*, которая сравнивает оба поля и, если они полностью совпадают, возвращает значение *True*:

```
//ПРОВЕРИТЬ, НЕ РЕШЕНА ЛИ ЗАДАЧА
function TForm1.IsReady: Boolean;
var
  i, j: integer;
begin
  Result:= True;
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do
      if masPole[i,j]<> masPole2[i,j] then
        begin Result:= False; exit end
end; //IsReady
```

Если вам повезёт, то программа поздравит вас с успехом надлежащей надписью. Хотелось бы миг победы сделать ещё более приятным, поэтому давайте *художественно* изобразим поздравление на отдельной форме *frmReady* (Рис. 4.10).



Рис. 4.10. Молодца!

```
AutoSize = True
BorderStyle = bsNone
Position = poDesktopCenter
```

Всю клиентскую часть занимает компонент *Image1: TImage*:

```
Left = 0
Top = 0
Width = 353
Height = 170
```

На нём установите *метку Label1*:

```
Left = 12
Top = 32
Width = 329
Height = 46
Caption = 'Вы сделали это!'
Font.Charset = RUSSIAN_CHARSET
Font.Color = clBlue
Font.Height = -40
Font.Name = 'Arial'
Font.Style = [fsBold]
Transparent = True
```

В *процедуре Ready* метка возвращается в исходное положение (скоро вы узнаете почему), форма с надписью показывается на экране и запускается таймер *Timer1*:

```
//ЗАДАЧА РЕШЕНА!
procedure TForm1.Ready;
begin
```

```
  rep:= 0;
  ReadyUnit.frmReady.label1.Top:= 32;
  frmReady.Show;
  timer1.Enabled:= true;
end; //Ready
```

Таймер (**Enabled** = *False*, **Interval** = *100*) периодически вызывает *процедуру OutReady*, где фон заполняется случайными точками серого и белого цветов:


```
//СОЗДАТЬ ФОН ДЛЯ НАДПИСИ
procedure OutReady;
var
  bmp: TBitmap;
  i, j: integer;
begin
  //создать временный растр:
  bmp:=TBitmap.Create;
  bmp.Height:= ReadyUnit.frmReady.Image1.Height;
  bmp.Width:= ReadyUnit.frmReady.Image1.Width;
  //заполнить его точками серого и белого цвета:
  for j:= 0 to bmp.Height do
    for i:= 0 to bmp.Width do
      if random(2) = 0 then bmp.Canvas.Pixels[i,j]:= clGray
      else bmp.Canvas.Pixels[i,j]:= clWhite;
  //обвести рамкой:
  bmp.Canvas.Brush.Style:= bsClear;
  bmp.Canvas.Pen.Width:= 5;
  bmp.Canvas.Pen.Color:= clYellow;
  bmp.Canvas.Rectangle(0,0,bmp.Width,bmp.Height);
  //скопировать в объект Image1:
  ReadyUnit.frmReady.Image1.Picture.Bitmap.Assign(bmp);
  //уничтожить растр:
  bmp.FreeImage;
end;  //OutReady;
```

На этот фон накладывается надпись с поздравлением, которая медленно перемещается вниз и время от времени меняет цвет (Рис. 4.11). После исчерпания заданного числа срабатывания таймер выключается, а форма убирается с экрана:

```
//ПОЗДРАВИТЬ
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  OutReady;
  inc(rep);
  if rep mod 10 > 5  then ReadyUnit.frmReady.label1.Font.Color:=
clBlue
  else ReadyUnit.frmReady.label1.Font.Color:= clRed;
  ReadyUnit.frmReady.label1.Top:= ReadyUnit.frmReady.label1.Top+1;
```

```
  if rep> 60 then begin
    timer1.Enabled:= false;
    frmReady.Hide;
  end;
end;  //Timer1Timer
```



Рис. 4.11. Художественная вещь!

Если вы захотите выставить позицию на игровом поле, то нажимайте *левую* кнопку мыши, удерживая клавишу *Shift*:

```
//ВЫСТАВИТЬ ФИШКУ ИЛИ СДЕЛАТЬ ХОД
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);
  if ACol * ARow = 0 then exit;
  //нажата левая кнопка мыши и клавиша Shift -
  //ставим след. фишку:
  if (ssLeft in shift) and (ssShift in shift ) then begin
    inc(masPole[ACol,ARow]);
    if masPole[ACol,ARow]>4 then masPole[ACol,ARow]:= 0;
    dgPole.Invalidate;
  end
  else if (ssLeft in shift) and (ssAlt in shift )then begin
    //не нужно ходить в пустую клетку:
    if masPole[ACol, ARow]= 0 then exit;
    //ход назад:
    masPole:=BackMove(masPole, ACol, ARow);
    inc(Hod); lblHod.Caption:= 'Ход - ' + inttostr(Hod);
    dgPole.Invalidate;
    //запомнить ход:
    Moves[Hod].Col:= ACol;
    Moves[Hod].Row:= ARow;
    Moves[Hod].PosL:= masPole;
    Hod_max:= Hod;
```

```
    //занести ход в протокол:
    frmProtokol.ListBox1.items.add(format('%3d',[Hod]) + '. ' +
                         letter[ACol]+ ' ' +inttostr(ARow));
  end
  //нажата левая кнопка мыши без клавиши Shift и Alt -
  //делаем ход:
  else if ssLeft in shift then begin
    if masPole[ACol, ARow]<> 0 then exit;
    masPole:=DoMove(masPole, ACol, ARow);
    inc(Hod); lblHod.Caption:= 'Ход - ' + inttostr(Hod);
    dgPole.Invalidate;
    //запомнить ход:
    Moves[Hod].Col:= ACol;
    Moves[Hod].Row:= ARow;
    Moves[Hod].PosL:= masPole;
    Hod_max:= Hod;
    //занести ход в протокол:
    frmProtokol.ListBox1.items.add(format('%3d',[Hod]) + '. ' +
                         letter[ACol]+ ' ' +inttostr(ARow));
    //проверить, не решена ли задача:
    if IsReady then Ready;
  end
end;  //dgPoleMouseDown
```


Дабы обеспечить себе возможность *возврата* к прежним позициям, мы запоминаем очередной ход и позицию на обоих полях в массиве *Moves*.

А о том, для чего нужно удерживать клавишу *Alt*, читайте дальше.

Всплывающее меню

Всплывающее меню в этой программе существенно уменьшилось в размерах по сравнению с игрой *ShortGame* (Рис. 4.12).

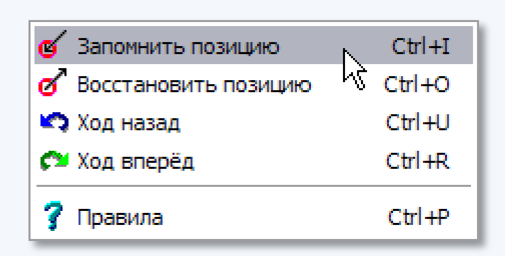


Рис. 4.12. Всплывающее меню

Вы даже можете совсем удалить его из проекта, заменив все оставшиеся пункты меню *кнопками* на форме, благо места для них теперь более чем достаточно.

Выбрав *первый* пункт меню, вы запомните игровую ситуацию

```
//ЗАПОМНИТЬ ПОЗИЦИЮ
procedure TForm1.MemoryIn;
begin
  Memory.hod:= Hod;
  Memory.PosL:= masPole;
  Memory.PosR:= masPole2;
end; //MemoryIn
procedure TForm1.miMemoryInClick(Sender: TObject);
begin
  MemoryIn;
end; //miMemoryInClick
```


А затем в любое время сможете вернуться к ней, щёлкнув на *втором* пункте:

```
//ВОССТАНОВИТЬ ПОЗИЦИЮ НА ПОЛЕ
procedure TForm1.miMemoryOutClick(Sender: TObject);
var i: integer;
begin
  with Memory do begin
    masPole:= PosL;
    masPole2:= PosR;
  end;
  dgPole.Invalidate;
  dgPole2.Invalidate;
  Hod:= Memory.hod;
  lblHod.Caption:= 'Ход - ' + inttostr(hod);
  //обновить протокол:
  frmProtokol.ListBox1.items.Clear;
  for i:= 1 to hod do
    frmProtokol.ListBox1.items.add(format('%3d',[i]) + '. ' +
                          letter[Moves[i].Col]+ ' '
+inttostr(Moves[i].Row));
end; //miMemoryOutClick
```

Следующий пункт меню вернёт вас к предыдущему ходу. А чтобы не возникало временн***ы***х коллизий, откат ограничен запомненной позицией:

```
//ВЕРНУТЬ ХОД
procedure TForm1.miUndoClick(Sender: TObject);
var i: integer;
begin
  if hod= Memory.Hod then exit;
  dec(hod);
  lblHod.Caption:=  'Ход - ' + inttostr(hod);
  masPole:= Moves[hod].PosL;
  dgPole.Invalidate;
  //обновить протокол:
  frmProtokol.ListBox1.items.Clear;
```

```
  for i:= 1 to hod do
    frmProtokol.ListBox1.items.add(format('%3d',[i]) + '. ' +
                             letter[Moves[i].Col]+         '          '
+inttostr(Moves[i].Row));
end; //miUndoClick
```

С той же лёгкостью вы сможете повторить отвергнутый ход:

```
//ОТМЕНИТЬ ПОСЛЕДНИЙ ВОЗВРАТ ХОДА
procedure TForm1.miRedoClick(Sender: TObject);
var i: integer;
begin
  if Hod< Hod_max then inc(hod)
  else exit;
  //восстановить позицию на игровом поле:
  masPole:=Moves[Hod].PosL ;
  dgPole.Invalidate;
  lblHod.Caption:= 'Ход - ' + inttostr(hod);
  //обновить протокол:
  frmProtokol.ListBox1.items.Clear;
  for i:= 1 to hod do
    frmProtokol.ListBox1.items.add(format('%3d',[i]) + '. ' +
                             letter[Moves[i].Col]+ ' '
+inttostr(Moves[i].Row));
end;  //miRedoClick
```


Последний пункт меню, показывающий справочный файл, мы рассмотрим после того, как закончим конструирование интерфейса программы.

Весьма удобной может оказаться возможность *обнулить* ходы. Для этого достаточно просто щёлкнуть на метке, показывающей число сделанных ходов:

```
//ОБНУЛИТЬ ХОДЫ
procedure TForm1.lblHodClick(Sender: TObject);
begin
  Hod:= 0;    Hod_max:= 0;
  lblHod.Caption:=  'Ход - 0';
  frmProtokol.listbox1.items.add('');
end;  //lblHodClick
```

«Отзывчивые» кнопки

Вы, должно быть, много раз видели в различных программах кнопки, которые изменяют свой вид, когда курсор мыши заходит на них, и возвращаются в прежнее состояние, когда он их покидает. Ничего не скажешь, эффектное зрелище! А между тем такие визуальные превращения нетрудно встроить и в свою собственную программу. Можно

даже использовать кнопки из программы *ShortGame*, если их чуточку доработать. Так мы и сделаем, а также добавим ещё несколько кнопок, необходимых для работы данной программы.

Все заранее подготовленные картинки в формате *BMP* загрузите в компоненты *TImage* на форме (Рис. 4.13).

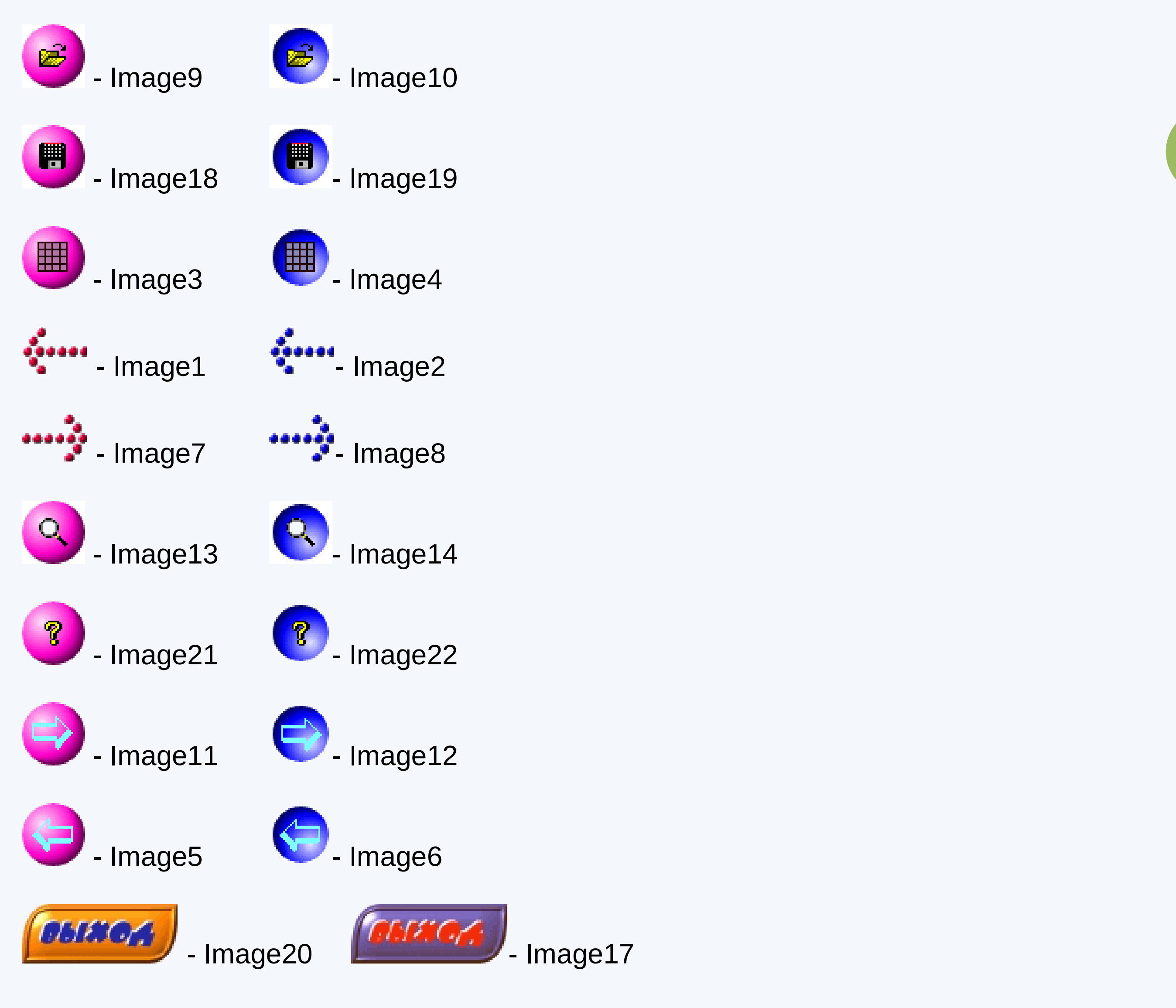


Рис. 4.13. Кнопочки программные!

Обратите внимание на то, что *розовые* кнопки немного больше по размеру, чем *синие* (величина всех картинок одинакова – 36 x 36 пикселей, чтобы центры кнопок в обоих положениях совпадали), поэтому кнопка под мышкой будет не только изменять цвет, но и «раздуваться».

Вот так выглядит форма с компонентами в окне *Конструктора формы* (Рис. 4.14).

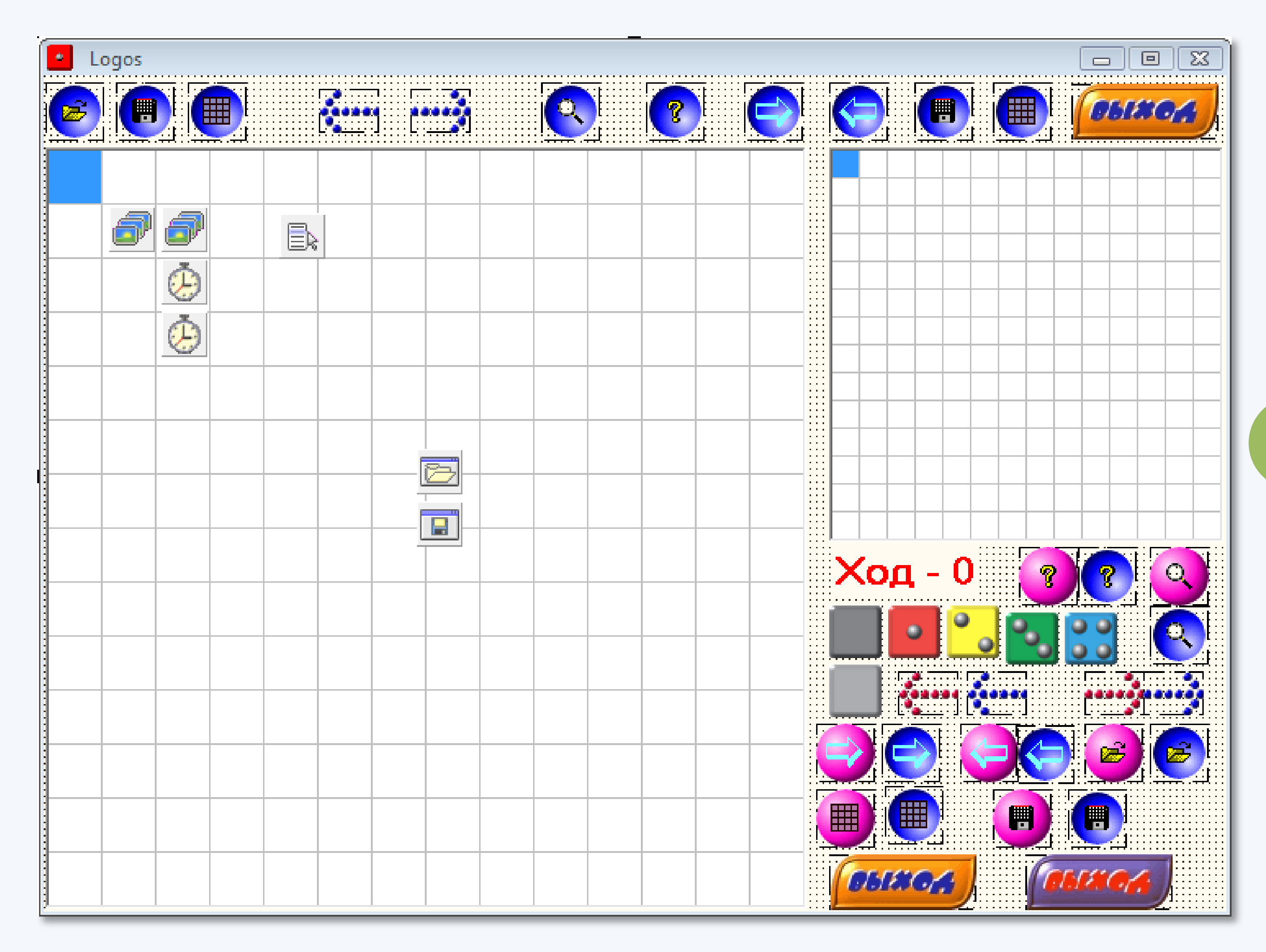


Рис. 4.14. Кнопочный боекомплект!

Как и в игре *ShortGame*, для каждой кнопки нужно установить на форме ещё один компонент *TImage* и загрузить в него ту картинку, которая соответствует неактивной (отжатой) кнопке. Большинство свойств у этих компонентов одинаковы (у «некруглых кнопок» другие размеры и верхняя координата):

```
Width = 36
Height = 36
Top = 4
AutoSize = True
Cursor = crHandPoint
ShowHint = True
Transparent = True
```

Различаются у них свойства *Hint* (подсказка), *Left, Top* (положение на форме) и *Tag* (по нему мы и будем определять, над какой кнопкой находится курсор мыши).

Рассмотрим действие самой простой кнопки – *Выход*. Она сделана в ином стиле, чем все остальные кнопки, чтобы по ошибке не «выпасть» из программы. Её изображение находится в компоненте *imgExit* (Рис. 4.15).



Рис. 4.15. Выход есть всегда!

```
Left = 628
Top = 5
Height = 34
Width = 89
Hint = 'Выход из программы'
Tag = 111

OnMouseDown = imgExitMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgExitMouseUp
```


При нажатии на эту кнопку возникает событие, которое обрабатывается *процедурой imgExitMouseDown*. В ней просто одна картинка *заменяется* другой:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.imgExitMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgExit.Picture.Assign(Image20.Picture);
end; //imgExitMouseDown
```

При отпускании кнопки возвращается прежняя картинка и программа закрывается:

```
procedure TForm1.imgExitMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgExit.Picture.Assign(Image17.Picture);
  close
end; //imgExitMouseUp
```

Но самые интересные превращения кнопки происходят тогда, когда на неё «наезжает» курсор. Вообще-то мы не можем определить момент захода курсора на кнопку и схода с неё, так как у компонента *TImage* нет соответствующего события (это верно для «старых» версий *Delphi*, в *Turbo* эти события имеются, но для совместимости мы поступим не самым

лучшим образом!). Поэтому придётся воспользоваться другим свойством - *MouseMove*, которое возникает при движении мыши по компоненту.

Так как кнопок у нас довольно много, то нужно добавить в проект более десятка похожих процедур. Хуже того – подобные процедуры потребуются и для других компонентов на форме и самой формы, иначе нам не удастся зафиксировать момент схода курсора с кнопки – ведь при этом не возникает никакого события, просто курсор начнёт перемещаться по другому компоненту. Правильнее будет всю обработку перемещений курсора перенести в одну процедуру, например *FormMouseMove*, в которой и определять, где находится курсор.

По умолчанию значение свойства *Tag* у всех компонентов (и у формы в том числе) равно *нулю*, поэтому, присвоив кнопкам другие, индивидуальные значения, мы сможем установить, какая кнопка (*Sender*) вызвала этот обработчик.

Но, в первую очередь, необходимо обеспечить прямое попадание в процедуру *FormMouseMove* при движении курсора на других компонентах. Поэтому для каждого компонента (кнопок и всех близлежащих) в *Инспекторе объектов*, на вкладке *Events* следует присвоить свойству *OnMouseMove* значение *FormMouseMove* (достаточно выбрать его из выпадающего списка). Вот теперь при заходе курсора на кнопки и сходе с них будет вызываться процедура *FormMouseMove*. По значению свойства *Tag* мы определяем, какой компонент находится под курсором. Но тут есть одна тонкость. Если курсор только что зашёл на кнопку, то картинка на ней изменилась и при дальнейшем движении её больше не нужно выводить на кнопку, иначе она будет слегка мигать.

Чтобы избежать этого, в глобальной *переменной*

```
LastComp: integer= 0;
```

мы будем хранить значение тега того компонента, который *перед этим* вызывал процедуру *FormMouseMove*. Тогда мы легко проверим, сменился компонент или нет. Если это тот же самый компонент, то мы просто завершаем процедуры, без всяких последствий для кнопки.

Ну а дальше всё просто. Если курсор заходит на «новую» кнопку, то картинка на ней изменяется, а значение тега запоминается в переменной *LastComp*. Если до этого курсор был на *другой* кнопке, то прежде её следует вернуть в неактивное состояние:

```
//МЫШКА ПЕРЕМЕЩАЕТСЯ ПО ФОРМЕ И КОМПОНЕНТАМ НА НЕЙ
```

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var n: integer;
begin
  //на каком компоненте находится курсор:
  n:= (Sender as TComponent).tag;
  //тот же компонент - ничего не делать:
  if n= LastComp then exit;

  //выключить прежнюю кнопку:
  case LastComp of
    101: imgSavePosL.Picture.Assign(Image19.Picture);
    102: imgClearL.Picture.Assign(Image4.Picture);
    103: imgSavePosR.Picture.Assign(Image19.Picture);
    104: imgOpen.Picture.Assign(Image10.Picture);
    105: imgClearR.Picture.Assign(Image4.Picture);
    106: imgSolution.Picture.Assign(Image14.Picture);
    107: imgCopyLR.Picture.Assign(Image12.Picture);
    108: imgCopyRL.Picture.Assign(Image6.Picture);
    109: imgUndo.Picture.Assign(Image2.Picture);
    110: imgRedo.Picture.Assign(Image8.Picture);
    111: imgExit.Picture.Assign(Image20.Picture);
    112: imgHelp.Picture.Assign(Image22.Picture);
  end;

  //зашли на новую кнопку - включить:
  LastComp:= (Sender as TComponent).tag;
  case LastComp of
      0: exit;  //- не кнопка
    101: imgSavePosL.Picture.Assign(Image18.Picture);
    102: imgClearL.Picture.Assign(Image3.Picture);
    103: imgSavePosR.Picture.Assign(Image18.Picture);
    104: imgOpen.Picture.Assign(Image9.Picture);
    105: imgClearR.Picture.Assign(Image3.Picture);
    106: imgSolution.Picture.Assign(Image13.Picture);
    107: imgCopyLR.Picture.Assign(Image11.Picture);
    108: imgCopyRL.Picture.Assign(Image5.Picture);
    109: imgUndo.Picture.Assign(Image1.Picture);
    110: imgRedo.Picture.Assign(Image7.Picture);
    111: imgExit.Picture.Assign(Image17.Picture);
    112: imgHelp.Picture.Assign(Image21.Picture);
  end;
end;  //FormMouseMove
```


Вы можете добавлять сколько угодно своих кнопок, не забывая при этом дописывать нужные строки в процедуру *FormMouseMove*. Так как изображения кнопок в активном и пассивном положениях могут быть любыми, то вам доступны всякие визуальные эффекты. Вот так простыми средствами можно получить вполне профессиональный интерфейс.

Но продолжим разбор кнопок.

Очень просты в программировании кнопки, которые дублируют пункты меню. Например, кнопки *imgUndo* и *imgRedo* вынесены на форму для удобства пользования, так как постоянно открывать меню – занятие довольно нервное. Они попросту вызывают те же процедуры, что и соответствующие пункты меню.

```
Left = 168
Top = 8
Height = 27
Width = 37
Hint = 'Ход назад'
Tag = 109
OnMouseDown = imgUndoMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgUndoMouseUp
```

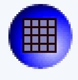
```
Left = 224
Hint = 'Ход вперёд'
Tag = 110
OnMouseDown = imgRedoMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgRedoMouseUp
```

```
//ВОЗВРАТ ХОДА
procedure TForm1.imgUndoMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgUndo.Picture.Assign(Image2.Picture);
end; //imgUndoMouseDown
procedure TForm1.imgUndoMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  miUndoClick(Self);
  imgUndo.Picture.Assign(Image1.Picture);
end; //imgUndoMouseUp

//ХОД ВПЕРЁД
procedure TForm1.imgRedoMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgRedo.Picture.Assign(Image8.Picture);
```

```
end; //imgRedoMouseDown
procedure TForm1.imgRedoMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  miRedoClick(Self);
  imgRedo.Picture.Assign(Image7.Picture);
end; //imgRedoMouseUp
```

Несложно и убрать «мусор» с левого и правого полей. В *первом* случае достаточно занести в массив *masPole* нули, а затем обновить поле. Во *втором* нужно обновить и правое, и левое поле, чтобы «синхронизировать» их содержимое.



imgClearL:

```
Left = 88
Top = 4
Hint = 'Очистить левое поле'
Tag = 102
OnMouseDown = imgClearLMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgClearLMouseUp
```

imgClearR:

```
Left = 580
Hint = 'Очистить правое поле'
Tag = 105
OnMouseDown = imgClearRMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgClearRMouseUp
```

```
//ОЧИСТИТЬ ЛЕВОЕ ПОЛЕ
procedure TForm1.imgClearLMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgClearL.Picture.Assign(Image4.Picture);
end; //imgClearLMouseDown
procedure TForm1.imgClearLMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do
```

```
      masPole[i,j]:= 0;
  dgPole.Invalidate;
  MemoryIn;
  imgClearL.Picture.Assign(Image3.Picture);
end; //imgClearLMouseUp

//ОЧИСТИТЬ ПРАВОЕ ПОЛЕ
procedure TForm1.imgClearRMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  imgClearR.Picture.Assign(Image4.Picture);
end; //imgClearRMouseDown
procedure TForm1.imgClearRMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var
  i, j: integer;
begin
  for j:= 1 to dgPole2.RowCount-1 do
    for i:= 1 to dgPole2.ColCount-1 do
      masPole2[i,j]:= 0;
  dgPole2.Invalidate;
  masPole:= masPole2;
  dgPole.Invalidate;
  MemoryIn;
  frmProtokol.listbox1.Clear;
  imgClearR.Picture.Assign(Image3.Picture);
end; //imgClearRMouseUp
```


Очень полезными при разработке собственных заданий могут оказаться следующие кнопки.

imgCopyLR:



```
Left = 428
Hint = 'Копировать в правое поле'
Tag = 107
OnMouseDown = imgCopyLRMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgCopyLRMouseUp
```

imgCopyRL:

```
Left = 480
Hint = 'Копировать в левое поле'
Tag = 108
```

```
OnMouseDown = imgCopyRLMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgCopyRLMouseUp
```

Нажатие на *первую* из них приводит к копированию фишек с левого поля в правое:

```
//КОПИРОВАТЬ ЛЕВОЕ ПОЛЕ В ПРАВОЕ
procedure TForm1.imgCopyLRMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  //заменить картинку:
  imgCopyLR.Picture.Assign(Image12.Picture);
  masPole2:= masPole;
  dgPole2.Invalidate;
end; //imgCopyLRMouseDown
procedure TForm1.imgCopyLRMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgCopyLR.Picture.Assign(Image11.Picture);
end; //imgCopyLRMouseUp
```

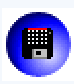

Нажатие на *вторую* приведёт к прямо противоположному результату:

```
//КОПИРОВАТЬ ПРАВОЕ ПОЛЕ В ЛЕВОЕ
procedure TForm1.imgCopyRLMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  imgCopyRL.Picture.Assign(Image6.Picture);
  masPole:= masPole2;
  dgPole.Invalidate;
end; //imgCopyRLMouseDown
procedure TForm1.imgCopyRLMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgCopyRL.Picture.Assign(Image5.Picture);
end; //imgCopyRLMouseUp
```

Решив предложенные задачи, вы наверняка захотите составить и свои. Тогда незамысловато выставляйте нужные фишки на одном из полей или делайте ходы на левом поле, а затем полученную позицию сохраните на диске. Таким образом вы сможете пополнить библиотеку заданий для этой игры.

Чтобы сохранить игровое поле, нажмите левую кнопку.

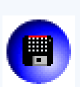
imgSavePosL:



```
Left = 44
Hint = 'Записать текущую позицию'
Tag = 101
OnMouseDown = imgSavePosLMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgSavePosLMouseUp
```


Чтобы сохранить поле-образец, такую же кнопку, но справа.

imgSavePosR:



```
Left = 532
Hint = 'Записать текущую позицию'
Tag = 103
OnMouseDown = imgSavePosLMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgSavePosLMouseUp
```

Так как и кнопки, и операции сохранения практически одинаковы для обоих полей, то давайте объединим их. Нажатие и отпускание кнопок осуществляется в *процедурах*

```
//ЗАПИСАТЬ ПОЗИЦИЮ НА ЛЕВОМ ИЛИ ПРАВОМ ПОЛЕ
procedure TForm1.imgSavePosLMouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSavePosL.Picture.Assign(Image19.Picture)
end; //imgSavePosLMouseDown
procedure TForm1.imgSavePosLMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSavePosL.Picture.Assign(Image18.Picture);
  if (Sender as TComponent).tag= 101 then begin // - левое поле
    SavePos(masPole) end
  else begin                                   // - правое поле
    SavePos(masPole2) end;
end; //imgSavePosLMouseUp
```

А непосредственное сохранение заданного поля происходит в процедуре *SavePos*. Все задачи записываются в папку *Figures* и имеют расширение *LGS*.

Причём сохраняется всё поле целиком, что существенно упрощает процедуры записи и загрузки заданий. И, наконец, в заголовке формы выводится имя файла (задачи), заданное вами при его сохранении. Одновременно это имя присваивается глобальной *переменной*

```
  NameFig: string= '';
```

Благодаря этому вы всегда легко определите название решаемой задачи. Если захотите, то можете расставить задачи по уровню сложности и после того, как пользователь вашей программы справится с одним заданием, предложить ему задание более сложное.


```
//ЗАПИСАТЬ ПОЗИЦИЮ НА ЗАДАННОМ ПОЛЕ
procedure TForm1.SavePos(arr: TPole);
var
  F: textfile;
  fn,s: string;
  i,j: integer;
begin
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Задачи (*.lgs)|*.LGS';
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'Figures\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите позицию на диск';
  if NameFig<>'' then savedialog1.filename:= NameFig
  else savedialog1.filename:='temp.lgs';
  if not savedialog1.Execute then exit;
  //имя конечного файла:
  fn:= savedialog1.filename;
  NameFig:=fn;
  assignfile(f,fn);
  rewrite(f);
  //записать позицию:
  for j:= 1 to dgPole.RowCount-1 do begin
    s:='';
    for i:= 1 to dgPole.ColCount-1 do s:=s+ inttostr(arr[i][j]);
    writeln (f,s);
  end;
  closefile(f);
  //вывести в заголовок формы имя файла:
  form1.caption:= NAME_PROG + '    [' + NameFig + ']';
  messagebeep(0)
end; //SavePos
```

Вполне естественно, что задачу на диске нужно уметь *загружать* для решения в правое поле. За это будет отвечать кнопка

imgOpen:

Left = 1
Hint = 'Загрузить задачу с диска'
Tag = 104
OnMouseDown = imgOpenMouseDown
OnMouseMove = FormMouseMove

```
//ЗАГРУЗИТЬ ЗАДАЧУ
procedure TForm1.imgOpenMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgOpen.Picture.Assign(Image10.Picture);
  Open('');
end; //imgOpenMouseDown
```


При нажатии этой кнопки вызывается процедура *Open* с *пустой* строкой в качестве параметра. В этом случае появляется диалоговое окно открытия файла, и вы можете выбрать любой из имеющихся на диске. Чтобы загрузить *определённую* задачу (например, при переходе на следующий уровень сложности), следует передать в процедуру *имя файла*. Осталось добавить, что если вы не планируете набирать условия задач в текстовом редакторе (а это очень легко сделать), то можно ещё более упростить процедуру, убрав из неё проверки правильности данных на диске.

```
//ЗАГРУЗИТЬ ЗАДАЧУ С ЗАДАННЫМ ИМЕНЕМ
procedure TForm1.Open (fn: string);
var
  s: string;
  F: TextFile;
  nLines,Len: integer;
  i,j:integer;
  ss: array[1..MAX_POLE_HEIGHT] of string;
begin
  if fn='' then begin  //- имя файла не задано
    form1.opendialog1.DefaultExt:='lgs';
    form1.opendialog1.Filter:='Logos files (*.lgs)|*.LGS';
    s:=extractfilepath(application.exename)+'FIGURES\';
    form1.opendialog1.InitialDir:= s;
    form1.opendialog1.Title:='Загрузите новую задачу';
    if not form1.opendialog1.Execute then exit;
    s:= form1.opendialog1.filename
    end
  else s:=extractfilepath(application.exename)+'FIGURES\'+ fn;

  //название задачи:
  NameFig:=s;

  {$i-}
```

```
  AssignFile(F,NameFig);
  Reset(F);
  {$i+}
  if IOResult<>0 then begin  //- ошибка при загрузке файла
    application.MessageBox ('Такой задачи нет!',NAME_PROG, MB_OK);
    exit
  end;

  nLines:=0;
  Len:=0;
  while not eof(f) do begin
    inc(nLines);
    //считать строку из файла:
    Readln(F, S);
    ss[nLines]:=s;
    If (Length(s) <> Len) and (Len<>0) Then begin
      application.MessageBox ('Неверная длина строки!',NAME_PROG,
MB_OK);
      exit
      end
    else
      Len:= Length(s);
  end;
  //закрыть файл:
  CloseFile(F);
  //проверить данные:
  if (Len> MAX_POLE_WIDTH) or (nLines> MAX_POLE_HEIGHT) then begin
    application.MessageBox('Неверные данные!',NAME_PROG, MB_OK);
    exit
  end;
  //вывести в заголовок имя файла:
  form1.caption:= NAME_PROG + '   [' + NameFig + ']';

  //очистить массивы полей:
  for j:= 1 to dgPole.RowCount-1 do
    for i:= 1 to dgPole.ColCount-1 do begin
      masPole[i,j]:= 0;
    end;
  //и протокол:
  frmProtokol.listbox1.items.Clear;
  //заполнить массив новыми данными:
  for j:=1 to nLines do
    for i:=1 to Len do
       masPole2[i][j]:= strtoint(ss[j,i]);
  //запомнить начальную позицию:
  MemoryIn;
  //обновить поля:
  dgPole.Invalidate;
  dgPole2.Invalidate;
  //обнулить ходы:
  Hod:= 0;   Hod_max:= Hod;
  lblHod.Caption:= 'Ход - ' + inttostr(Hod);
```

```
end; //Open
```

Очень удобно с помощью этой процедуры при открытии программы загружать задание первого уровня сложности. Это можно сделать в уже рассмотренной нами процедуре *FormShow:*

```
//ПОКАЗАТЬ ФОРМУ
procedure TForm1.FormShow(Sender: TObject);
begin
  open('001.lgs');
  frmProtokol.show;
end; //FormShow
```


В результате сразу же после старта программы в правое поле будет загружена задача, и пользователь сможет приступить к её решению.

Следует также позаботиться о сохранении найденного *решения* задачи. Оно находится в списке *ListBox1* на форме протокола. При щелчке на кнопке *Записать*, формируется название файла решения. Если задача уже имеет название (оно находится в заголовке формы, как вы знаете), то к нему добавляются буквы *otv*, чтобы задача и её решение имели схожие имена. Если задача *безымянная*, то ответ будет записан в файл *otv_temp.txt* (файл ответа (решения) имеет расширение *txt*, так как это обычный текстовый файл):

```
//ЗАПИСАТЬ ОТВЕТ
procedure TfrmProtokol.sbtSaveOtvClick(Sender: TObject);
var
  s,ss,fn: string;
  i: integer;
  f: textfile;
begin
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Решения (*.txt)|*.TXT';
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'Figures\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите решение на диск';

  //сформировать имя файла:
  if NameFig<>'' then begin
    s:=extractFileName(NameFig);
    ss:='';
    i:=1;
    while (s[i]<>'.') and (i<=length(s)) do begin
      ss:=ss+s[i];
      inc(i)
    end;
    ss:=ss+'otv.txt';
```

```
    savedialog1.filename:=ss
   end
   else
     savedialog1.filename:='otv_temp.txt';

   if not savedialog1.Execute then exit;
   //имя конечного файла:
   fn:= savedialog1.filename;
   assignfile(f,fn);
   rewrite(f);
   //записать протокол:
   for i:=0 to (ListBox1.Items.Count - 1) do
     writeln (f,ListBox1.Items.Strings[i]);
   Closefile(f);
   Messagebeep(0)
end; //sbtSaveOtvClick
```


Очистить протокол вы сможете, нажав кнопку *Стереть*:

```
//ОЧИСТИТЬ ПРОТОКОЛ
procedure TfrmProtokol.SpeedButton1Click(Sender: TObject);
begin
  ListBox1.Items.Clear;
end; //SpeedButton1Click
```

Веб спешит на помощь

Теперь, когда разработка интерфейса программы почти завершена (автоматическое решение задачи будет рассмотрено в следующем разделе), мы можем написать «справочное пособие» по игре *Logos*. Как вы знаете, это можно сделать по-разному, но в этой игре мы освоим новый способ – оформим файл справки в виде веб-страницы. *Internet Explorer* уже интегрирован в операционную систему *Windows*, так что проблем с просмотром таких страниц ни у кого не будет. А, кроме новизны, мы получим современную справку, которую легко разработать хотя бы в текстовом редакторе *MS Word* (вы также легко найдёте множество программ, специально предназначенных для создания веб-страниц и веб-дизайна). Простую веб-страницу сделать ничуть не сложнее, а, пожалуй, даже и легче, чем справку в формате *HLP*.

Итак, кнопка *imgHelp*



```
Left = 368
```

```
Hint = 'Помощь'
Tag = 112
OnMouseDown = imgHelpMouseDown
OnMouseMove = FormMouseMove
OnMouseUp = imgHelpMouseUp
```

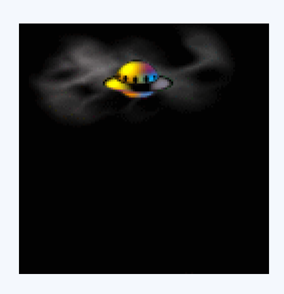
и пункт меню *miHelp* должны запустить *Internet Explorer* и открыть в нём наш файл справки. Для этого мы воспользуемся функцией *ShellExecute* из *Windows API* с соответствующими параметрами. Но прежде следует добавить модуль *ShellApi* в раздел *uses* (он находится в начале модуля главной формы *LogosUnit*).

Вызов справки совершенно одинаков в обоих случаях, поэтому вы можете объединить две следующие процедуры в одну.

```
//ПОКАЗАТЬ ФАЙЛ СПРАВКИ
procedure TForm1.miHelpClick(Sender: TObject);
var s: string;
begin
  s:= extractfilepath(application.exename)+'LogosHelp.htm';
  ShellExecute(handle, pChar('open'),pChar('iexplore'),pChar(s),
  '', SW_SHOWMAXIMIZED);
end;  //miHelpClick
procedure TForm1.imgHelpMouseDown(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
begin
  imgHelp.Picture.Assign(Image22.Picture);
end;  //imgHelpMouseDown
procedure TForm1.imgHelpMouseUp(Sender: TObject; Button: TMouse-
Button;
  Shift: TShiftState; X, Y: Integer);
var s: string;
begin
  imgHelp.Picture.Assign(Image21.Picture);
  s:= extractfilepath(application.exename)+'LogosHelp.htm';
  ShellExecute(handle, 'open','iexplore',pChar(s),
  '', SW_SHOWMAXIMIZED);
end;  //imgHelpMouseDown
```

Дело за малым – написать файл справки в редакторе *MS Word* и сохранить его как веб-страницу (вы также можете подобрать *готовую* веб-страницу и изменить надлежащим образом текст и рисунки на ней).

Файл справки может быть таким:

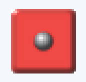
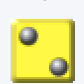
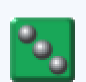
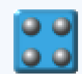


LOGOS

Отличная игра-головоломка, суть которой состоит в том, чтобы составить на игровом поле заданную фигуру из фишек.

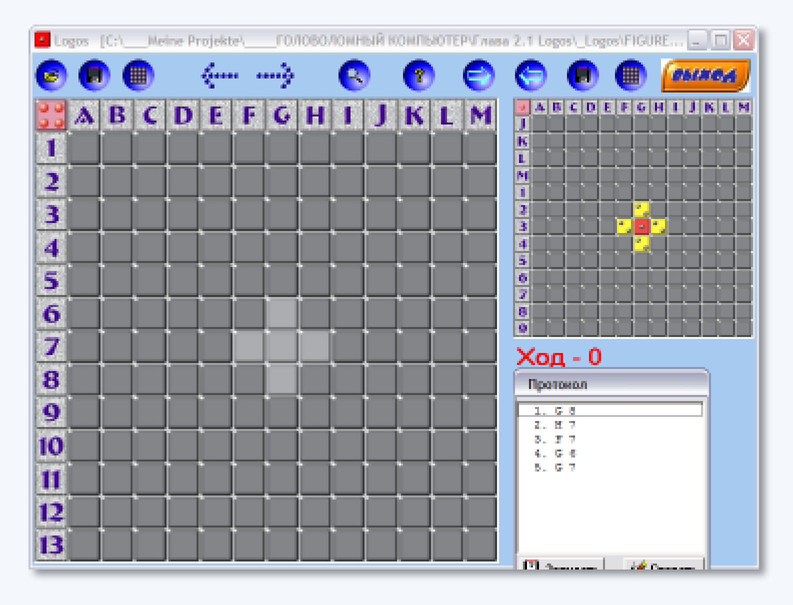
Правила игры

Каждая фишка имеет на верхней стороне 1, 2, 3 или 4 очка:

Фишки последовательно выставляются на **свободные клетки** игрового поля. У новой фишки на верхней стороне всегда находится одно очко. А вот все соседние с ней по горизонтали (слева и справа) и вертикали (сверху и снизу) фишки увеличивают своё значение на единицу, то есть фишка с одним очком превращается в фишку с двумя очками, фишка с двумя очками – в фишку с тремя очками, фишка с тремя очками – в фишку с четырьмя очками. Так как фишек с пятью очками нет, то фишка с четырьмя очками вновь становится «одноочковой».

Искомая конфигурация фишек приведена на маленьком поле справа. На игровом поле выделены более светлым оттенком клетки для выполнения ходов.



Чтобы сделать *ход*, достаточно просто щёлкнуть мышкой в нужной клетке игрового поля. Например, для решения этой задачи необходимо сначала, в любой последовательности, щёлкнуть на внешних клетках G6, F7, H7, G8, а затем в центральной – G7. Ход «партии» вы можете видеть в плавающем окне с заголовком «Протокол».

Управление игрой

Управлять игрой можно с помощью ***кнопок*** в верхней части окна приложения или ***всплывающего меню***, которое вызывается нажатием правой кнопки мыши.

Управляющие кнопки



- открыть список заданий на диске; вы можете выбрать и загрузить любое из них.

- сохранить позицию на игровом поле (левая кнопка) или поле с заданием (правая кнопка). Позволяет разрабатывать и записывать на диск свои собственные задачи.

- стереть все фишки с игрового поля (левая кнопка) или с поля-образца (правая кнопка).

- отменить последний ход.

- вернуть отменённый ход.

- найти решение задачи.

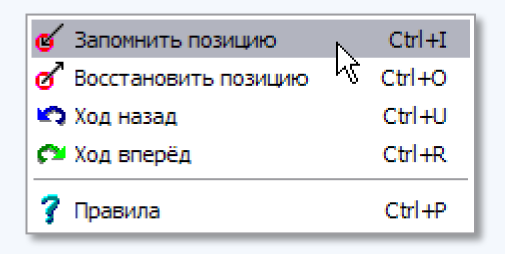
- показать справку помощи.

- копировать левое поле в правое.

- копировать правое поле в левое.

- закрыть программу.

Всплывающее меню



Последние три пункта меню просто дублируют соответствующие управляющие кнопки.

Выбрав первый пункт, вы запомните позицию на игровом поле, а, выбрав второй, восстановите её. Эти пункты меню помогут вам вернуться к тому положению на поле, решение которого вы уже нашли.

Как вы видите (Рис. 4.16), можно использовать различные шрифты и вставлять картинки, например, анимированные *GIF*-файлы; один из них находится в начале текста справки (летающая тарелка; естественно, в текстовом редакторе виден только один «кадр» анимации, так что зрелище, конечно, не то).

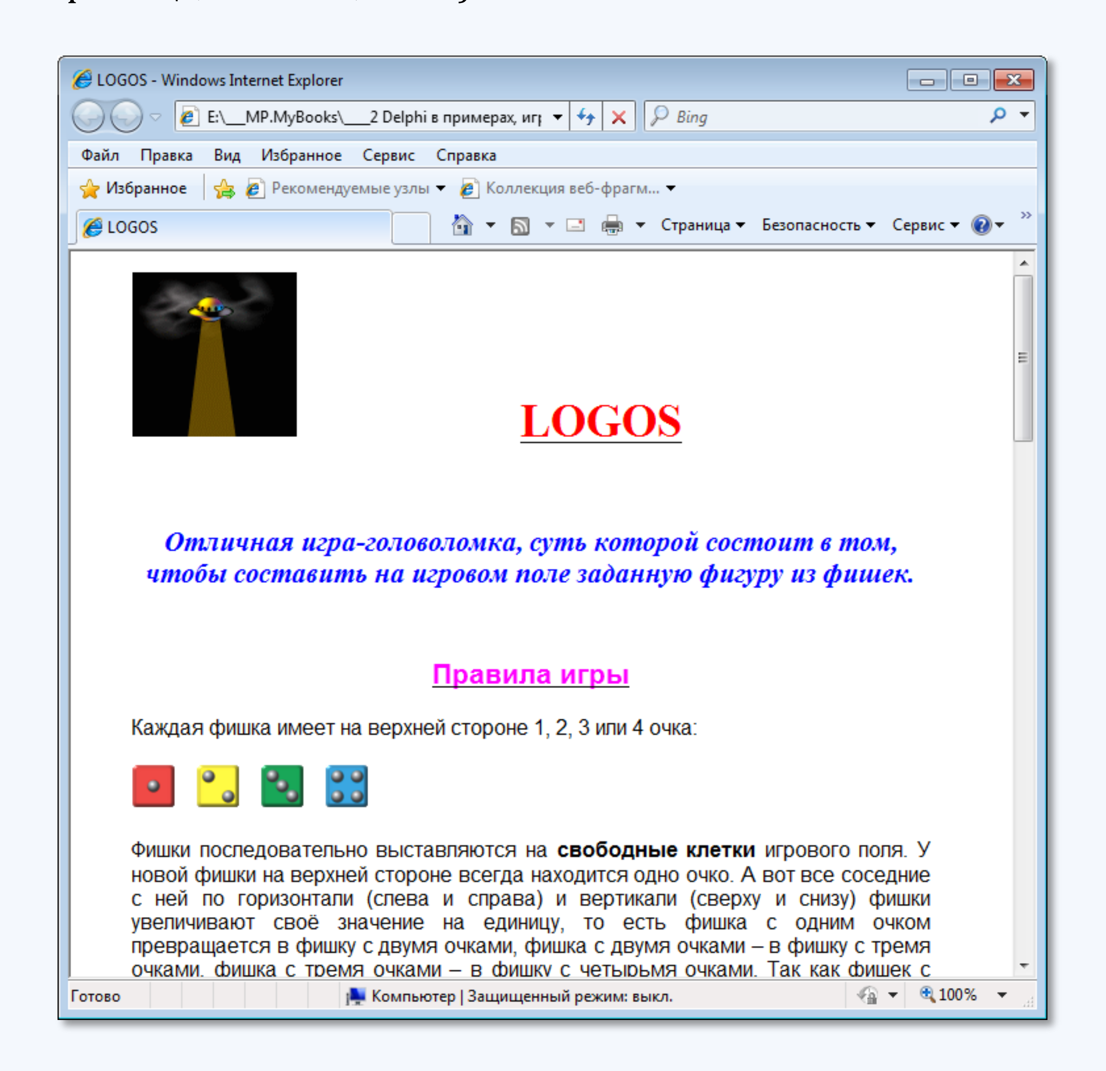


Рис. 4.16. Справка открыта в браузере *Internet Explorer*

Для разработки статичных картинок годится любой графический редактор, а для создания анимации существует много специализированных *GIF-аниматоров*, которые позволяют легко добавлять к вашим картинкам разнообразные визуальные спецэффекты.

После того как вы набрали текст помощи и вставили рисунки, добавьте переходы в нужные места файла. Для этого выделите текст, при щелчке на котором будет осуществляться переход, и вызовите контекстное меню, нажав *правую* кнопку мыши. В нём выберите пункт *Гиперссылка*, после чего откроется диалоговое окно *Вставка гиперссылки*. Так как место перехода находится в том же файле, то выберите слева пункт *Связать с местом в документе* и в списке укажите заголовок в файле помощи (у нас они выделены сиреневым цветом и подчёркиванием – не забудьте при наборе заголовка в списке стилей выбрать *Заголовок 1 (2, 3), начало документа* или *закладку!* Более подробно о создании гиперссылок вы можете прочитать в *Справке по Microsoft Office Word*.

И последнее – сохраните свою работу как веб-страницу в папке с игрой под именем *LogosHelp*. Для проверки загрузите справку в *Internet Explorer*, а если обнаружите недостатки, то прямо из него перешлите для доработки в *Word*. О том, как работать в редакторе, вы уже знаете.

Естественно, ту же самую справку вы увидите, если попросите помощи непосредственно из программы *Logos*. Я надеюсь, результаты вас полностью удовлетворят. Впрочем, вы можете сколько угодно улучшать свою «веб-страницу», украшая её новыми элементами. Но и в «простом» варианте помощь выглядит отлично!

Некоторые особенности программы не отражены в предлагаемом файле справки. Если вы считаете нужным упомянуть о них, то сделайте это!

Шиворот-навыворот и задом наперёд

Конец – делу венец.

Русская пословица

Если вы пробовали решать предложенные в начале главы задачи, то, конечно, ощутили на себе, как непросто решить даже эти элементарные «примеры». Что уж говорить о «навороченных»!

А всё потому, что большинство фишек в ходе игры претерпевает несколько превращений, за которыми трудно уследить. Те, кто играл в реверси, прекрасно знакомы с подобной проблемой.

Безусловно, по конечной позиции сложно вычислить, с какой фишки началась игра и какие фишки пристраивались к ней потом. Зато с *последней* фишкой всё гораздо проще – из правил игры неотвратимо следует, что последняя фишка обязательно имеет *одно* очко на верхней грани, ведь после неё фишки больше не выставлялись, а, значит, она не могла изменить своего значения. Таким образом, вся «хитрость» задач игры *Logos* заключается в том, что их нужно решать с *конца*.

С ретроспективным анализом мы уже встречались при разработке игры *XorGame*, так что эта новость не должна застигнуть вас врасплох. Поэтому давайте решим с помощью этой методы задачи-примеры.

Задача 1.

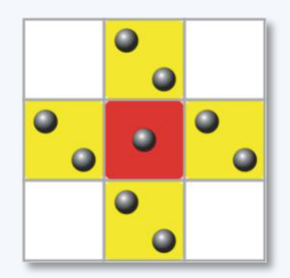


Рис. 4.17. Если не знаешь, с чего начать, начни с конца!

В конечной позиции только одна фишка – центральная – имеет одно очко на верхней грани (Рис. 4.17). Из вышесказанного совершенно очевидно, что она и была выставлена на поле последней.

С ней граничат 4 фишки, которые в результате последнего хода *увеличили* своё значение на единицу. То есть, если убрать центральную фишку, то все остальные фишки должны *уменьшить* своё значение на единицу. Стало быть, перед последним ходом ситуация на поле выглядела так (Рис. 4.18).

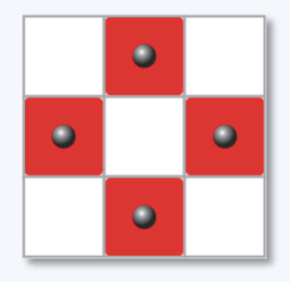


Рис. 4.18. Ход назад!

У этих фишек нет соседей, поэтому в ходе игры они не могли изменить своего значения, следовательно, их можно выставлять на поле в *любом* порядке, например, так (Рис. 4.19).

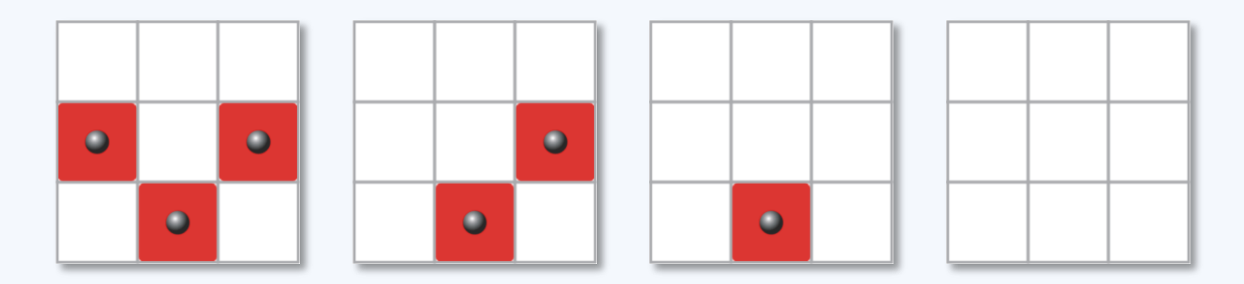


Рис. 4.19. Пятимся назад!

Таким образом, пользуясь ретроспективным анализом, мы пришли от конечной позиции к *начальной*. А в игре нужно, наоборот, перевести начальную позицию в конечную, поэтому следует *обратить* ходы - выставить сначала все периферийные фишки, а уже затем центральную.

Решим более сложную *задачу 2* (Рис. 4.20).

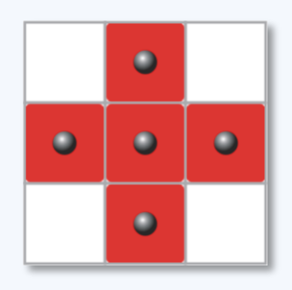


Рис. 4.20. Вторая задача

В конечной позиции *все* фишки имеют одно очко на верхней грани - так какая же из них была выставлена последней?

Легко понять, что фишка с одним очком либо вообще не изменяла своего значения в игре, либо изменяла его четырежды, чтобы совершить полное превращение: 1 → 2 → 3 → 4 → 1. Во втором случае у неё должно быть ровно 4 соседа. Отсюда следует важный вывод: *все фишки с одним очком, имеющие не более трёх соседних, никогда не изменяли своего значения*, то есть после них на поле не выставлялись фишки, граничащие с ними по длине стороны.

Поскольку у всех периферийных фишек только *один* сосед, то их можно смело удалять с поля в любой последовательности (Рис. 4.21).

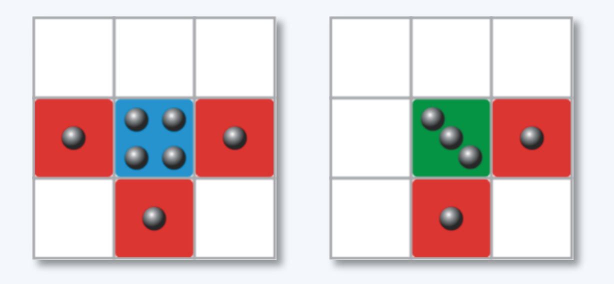


Рис. 4.21. Удаляем беспощадно



Дальнейшее решение не требует иллюстраций. Без сомнения, мы придём к центральной фишке с одним очком, которой и должен быть сделан первый ход.

В задаче 3 (Рис. 4.22) легко отыскать фишки с одним очком, имеющие менее четырёх соседей, - это все периферийные фишки.

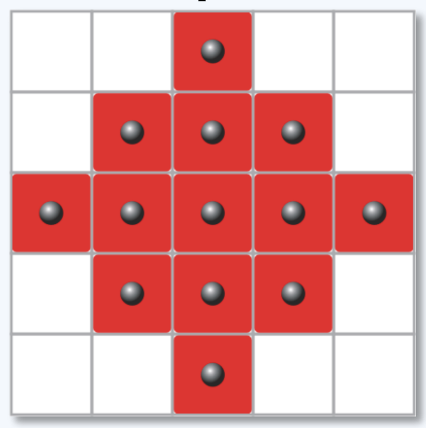


Рис. 4.22. Однако, ромбик!

Убирая их в любой последовательности (внимательно следите за превращениями остальных фишек!), мы неизбежно получим такую фигуру (Рис. 4.23).

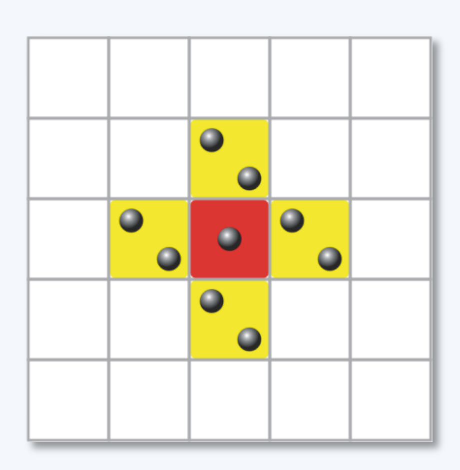


Рис. 4.23. Знакомая позиция!

Нетрудно узнать в ней первую задачу, которую мы уже благополучно решили.

Нечасто, но бывают такие ситуации на поле, когда *все* фишки с одним очком имеют по четыре соседа, поэтому найденное нами правило о трёх соседях здесь не действует. Например, в *задаче 25* после удаления четырёх фишек на границе фигуры возникнет такая «неразрешимая» позиция (Рис. 4.24).

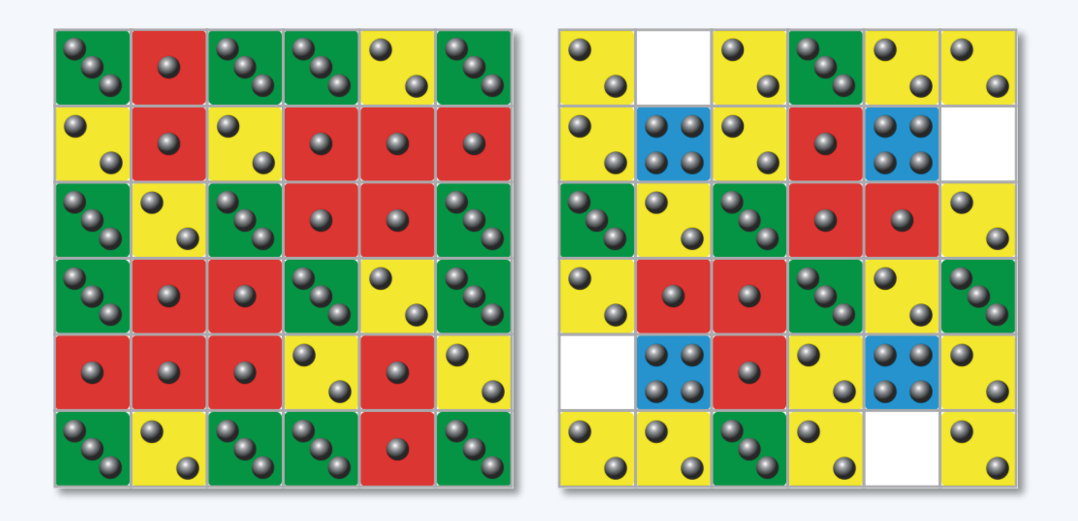


Рис. 4.24. Куда пойти?

Политически неверно было бы удалить здесь те фишки с одним очком, у которых нет соседних фишек с двумя очками. В результате таких ходов вы загоните себя в тупик, так как на поле вообще не останется ни одной фишки с одним очком и ходить будет нечем. Отсюда вытекает *второе* важное правило: *если на поле остались только такие фишки с одним очком, которые граничат с четырьмя другими, то удалять следует те из них, у которых есть соседняя фишка с двумя очками*.

Этих двух правил вполне достаточно, чтобы решить задачу любой сложности. Одна беда – при решении задачи мы будем двигаться в обратную сторону. Практически это выглядит так. Загружаем (или набираем вручную) в правое поле задачу, копируем её в левое поле кнопкой и последовательно удаляем фишки, пользуясь нашими правилами. Если вы нигде не собьётесь с верного пути, то полностью очистите игровое поле от фишек. Теперь нужно сделать то же самое, но в обратном порядке. Ходы запомнили? Шутка – у нас все ходы записаны в протоколе, поэтому просто ходите в нужные клетки, начиная с конца списка ходов. Небось, надоело высчитывать клетки по их координатам? Тогда щёлкните в списке на нужной строке – и фишка появится в нужном месте в нужное время. Щёлкнули, а она не появилась? И это верно: прежде

чем щёлкать, нужно подумать о последствиях, всё-таки мышка – это вам не волшебная палочка!

В модуле *ProtokolUnit* формы протокола нужно добавить *процедуру*, обрабатывающую щелчок на одной из строк списка. Чтобы не прокручивать длинный список ходов вручную,мы также поручим это нашей процедуре. Далее в ней «расшифровывается» строка с записью хода, после чего выполняется ход в нужную клетку игрового поля.


```
procedure TfrmProtokol.ListBox1Click(Sender: TObject);
var
  s,h: string;
  n: integer;
  x, y: integer;
begin
  //прокрутить список так, чтобы сверху была выделенная строка:
  ListBox1.TopIndex:= ListBox1.ItemIndex;
  //выделенная строка из списка:
  s:= ListBox1.Items.Strings[ListBox1.ItemIndex];
  //найти точку:
  n:= pos('.',s);

  if n=0 then exit; //- это не запись хода!

  //выделить номер хода:
  h:= copy(s,1,n-1);
  Hod:= strtoint(h);
  //вывести его на экран:
  LogosUnit.Form1.lblHod.Caption:= 'Ход - ' + inttostr(Hod);
  //выделить координаты клетки:
  x:= pos(s[n+2],letter);
  s:= copy(s,n+4,2);
  y:= strtoint (s);
  //выполнить ход:
  masPole:= LogosUnit.Form1.DoMove(masPole, x, y);
  Form1.dgPole.invalidate;
  //проверить, не решена ли задача:
  if LogosUnit.Form1.IsReady then LogosUnit.Form1.Ready;
end; //ListBox1Click
```

При этом решение задачи всё равно придётся начинать с *последних* ходов и подниматься в списке наверх. Было бы гораздо удобнее выполнять ходы «нормально», то есть сверху вниз. Для осуществления своих народных чаяний установим на форме протокола внизу большую *кнопку sbtInv*:

```
Left = 4     Top = 192
Width = 185     Height = 22
Hint = 'Обратить ходы'
Caption = 'Обратить ходы'
Font.Style = [fsBold]
```

OnClick = sbtInvClick

и наделим её способностью всё ставить с головы на ноги:

```
//ОБРАТИТЬ СПИСОК ХОДОВ
procedure TfrmProtokol.sbtInvClick(Sender: TObject);
var
  s: array [0..169] of string;
  str: string;
  i,j: integer;
  n,m: integer;
begin
  n:= ListBox1.Items.Count-1;

  j:=-1;
  for i:= 0 to n do
    if ListBox1.Items.Strings[i] <> '' then begin
      inc(j);
      s[j]:= ListBox1.Items.Strings[i];
  end;
  ListBox1.Clear;

  for i:= j downto 0 do begin
    m:= pos('.',s[i]);
    str:= copy(s[i], m+2, length(s[i])-m);
    str:= format('%3d',[j-i+1]) + '. ' + str;
    ListBox1.Items.Add(str);
  end;
  ListBox1.Items.Add('');
end; //sbtInvClick
```


После этих «ухищрений» вы сначала решаете задачу «ретроспективно», а затем обращаете последовательность ходов, щёлкаете на строках протокола и наблюдаете любопытные картины из жизни фишек на игровом поле.

Если вы внимательный читатель, то должны были заметить, что на самом деле вы не можете делать ходы на игровом поле в обратном направлении (убирать фишки), так как программа умеет только *выставлять* фишки на поле. Правда, можно вдавить поглубже клавишу *Shitf* и просто изменить нужные фишки так, чтобы создавалась иллюзия обратного хода, но всё иллюзорное так ненадёжно! Поэтому в процедуре *dgPoleMouseDown* (об этом говорилось выше) предусмотрена возможность делать ходы назад. Для этого надо щёлкать на нужной фишке, удерживая клавишу *Alt*. Тогда вместо функции *DoMove* вызывается функция *BackMove*, которая корректно снимает фишку с поля.

Разобраться в действии этой функции несложно, особенно если сравнить её с *DoMove*:

```
//ХОД НАЗАД
function TForm1.BackMove(arr: TPole; x, y: integer): TPole;
var i: integer;
begin
  result:= arr;
  //изменить соседние фишки:
  for i:= Max(y-1,1) to Min(y+1,dgPole.RowCount-1) do //- верти-
кальный
  begin
    if (i<> y) and (arr[x,i] <> 0) then begin
      dec(arr[x,i]);
      if arr[x,i]=0 then arr[x,i]:= 4; end;
  end;
  for i:= Max(x-1,1) to Min(x+1,dgPole.ColCount-1) do //- горизон-
тальный
  begin
    if (i<> x) and (arr[i,y] <> 0) then begin
      dec(arr[i,y]);
      if arr[i,y]=0 then arr[i,y]:= 4; end;
  end;
  //уменьшить на единицу значение заданной фишки:
  dec(arr[x,y]);
  result:= arr;
end; //BackMove
```


Учимся лениться

Не появилось ли у вас «естественного» желания вообще всю работу по решению задачи свести к одному щелчку на желанной кнопке ? Вопрос, конечно, риторический - но прежде чем предаться лени и неге, нужно взвалить всю нетворческую работу на плечи компьютера.

Нажать на кнопку мы, правда, всегда сумеем:

```
//НАЙТИ РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.imgSolutionMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  imgSolution.Picture.Assign(Image14.Picture);
end; //imgSolutionMouseDown
procedure TForm1.imgSolutionMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  imgSolution.Picture.Assign(Image13.Picture);
```

```
  if GameState= gsSolution then exit;
  //перевести программу в режим поиска:
  GameState:= gsSolution;
  //искать решение:
  solution(masPole2);
  //перевести программу в режим ожидания ввода команд:
  GameState:= gsWait;
end; //imgSolutionMouseUp
```

А вот с процедурой поиска решения, увы, так просто не разделаться. Хотя самое главное мы уже знаем – это наши правила решения задач, остальное, как говорится, дело техники.

Так, мы легко можем определить необходимое число ходов, ведь оно в точности равно числу фишек в конечной позиции. Затем отыскиваем в текущей позиции, пока можно, фишки с одним очком на верхней грани, имеющие не более трёх соседей (*Правило 1*, функция *Sosedi*). Когда такие фишки закончатся, переходим к поиску фишек с одним очком, имеющим четырёх соседей, в том числе с двумя очками на верхней грани. И так далее, и так далее, и так далее... В конце концов, либо задача будет решена, либо мы определим, что она неразрешима.

Естественно, задача и в этом случае решается в обратном порядке, но в процедуре *SavePos* ходы записываются как следует, так что вы можете пощёлкать на них в списке и убедиться в правильности решения задачи, а также и сохранить его на диске.

```
//АВТОМАТИЧЕСКОЕ РЕШЕНИЕ ЗАДАЧИ
procedure TForm1.Solution(arr: TPole);
var
  Hod: integer;
  w,h: integer;
  x, y: byte;
  flgHod: Boolean;
  arr1, arr2: TPole;

  //Подсчитать число ходов (фишек)
  function Steps: integer;
  var
    i, j: integer;
  begin
    Result:= 0;
    for j:= 1 to h do
      for i:= 1 to w do
        if arr2[i,j]<> 0 then inc(Result)
  end;

  //Подсчитать число соседей у единицы
  function Sosedi: integer;
```

```
var
  i, j: integer;
  i1: integer;
begin
  //искать по всему полю:
  for j:= 1 to h do
    for i:= 1 to w do
      if arr2[i,j]=1 then begin  //- нашли единицу
        Result:= 0;
        //сколько соседей?
        for i1:= Max(j-1,1) to Min(j+1,h) do //- по вертикали
          if (i1<> j) and (arr2[i,i1] <> 0) then begin //- есть
            inc(Result); end;
        for i1:= Max(i-1,1) to Min(i+1,w) do //- по горизонтали
          if (i1<> i) and (arr2[i1,j] <> 0) then begin //- есть
            inc(Result); end;
        if Result< 4 then begin
          x:= i; y:= j;
          //дальше не искать:
          exit;
        end;
      end;
  //не найдены:
  Result:= -1;
end;

//Найти единицу с 4-мя соседями (с двойкой)
function Sosedi2: integer;
var
  i, j: integer;
  i1: integer;
  flg: Boolean;
begin
  //искать по всему полю:
  for j:= 1 to h do
    for i:= 1 to w do
      if arr2[i,j]=1 then begin  //- нашли единицу
        Result:= 0;
        flg:= False;
        //сколько соседей?
        for i1:= Max(j-1,1) to Min(j+1,h) do //- по вертикали
          if (i1<> j) and (arr2[i,i1] <> 0) then begin //- есть
            inc(Result);
            if arr2[i,i1] =2 then flg:= True; end;
        for i1:= Max(i-1,1) to Min(i+1,w) do //- по горизонтали
          if (i1<> i) and (arr2[i1,j] <> 0) then begin //- есть
            inc(Result);
            if arr2[i1,j]= 2 then flg:= True; end;

        if flg and (Result=4) then begin  //- нашли
          x:= i; y:= j;
          //дальше не искать:
          exit; end;
```

```
        end;
    //не найдены:
    Result:= -1;
  end;

  //записать ходы в протокол:
  procedure SavePos;
  var
    n: integer;
    x, y: integer;
    s: string;
  begin
    frmProtokol.listbox1.items.Clear;
    for n:= Hod  downto 1 do begin
      s:='';
      x:= Move[n].x; y:= Move[n].y;
      frmProtokol.ListBox1.items.add(format('%3d',[Hod-n+1]) + '.
' +
                           letter[x]+ ' ' +inttostr(y));
    end;
    frmProtokol.listbox1.items.add('');
  end;

begin
  //очистить протокол:
  frmProtokol.listbox1.Clear;

  //скопировать массивы полей:
  arr2:= masPole2;
  arr1:= masPole;
  //размеры полей:
  w:= dgPole.ColCount-1; h:= dgPole.RowCount-1;

  //подсчитать число ходов:
  step:= steps;
  if step= 0 then begin
    application.MessageBox ('Вы не загрузили задачу!',NAME_PROG,
MB_OK);
    exit
  end;

  Hod:= 0;

  repeat
    //ходим в клетки с 1, имеющей не более 3 соседей:
    while Sosedi<> -1 do
    begin
      //делаем ход в эту клетку:
      inc(Hod);
      Move[Hod].x:= x;
      Move[Hod].y:= y;
      arr2:= BackMove(arr2, x, y);
      masPole:= arr2; dgPole.Invalidate;
```

```
      //делаем задержку, чтобы увидеть процесс решения задачи:
      sleep(100);
      application.ProcessMessages;
      if hod = step then begin
         savepos;
         frmProtokol.Hide;
         application.MessageBox ('Задача решена!',NAME_PROG,
MB_OK);
         frmProtokol.Show;
         exit
      end
    end;
    flgHod:= False;
    if Sosedi2<> -1 then begin
      //делаем ход в эту клетку:
      flgHod:= True;
      inc(Hod);
      Move[Hod].x:= x;
      Move[Hod].y:= y;
      arr2:= BackMove(arr2, x, y);
      masPole:= arr2; dgPole.Invalidate;
      sleep(100);
      application.ProcessMessages;
      if hod = step then begin
         savepos;
         frmProtokol.Hide;
         application.MessageBox ('Задача решена!',NAME_PROG,
MB_OK);
         frmProtokol.Show;
         exit
      end;
    end;
  until flgHod= False;
  //не удалось решить задачу:
  application.MessageBox ('Задача решений не имеет!',NAME_PROG,
MB_OK);
  exit;
end;  //Solution
```


Хотел бы предостеречь вас от бездумного использования автоматического решения задач, ведь в таком случае программа теряет всякий смысл! А вот при разработке и проверке собственных заданий кнопка вам пригодится для проверки своих творений. Не говоря уже о том, что найденные нами алгоритмы могут пригодиться вам при создании других программ!



Исходный код программы находится в папке **Logos**.



«С любимыми не расставайтесь…»

Если вы ещё не наигрались в *Logos*, то попробуйте приумножить достоинства этой игры новыми возможностями.

1. Если вы произвольно выставите фишки на поле, то, скорее всего, получите неразрешимую задачу. Тогда можно постараться решить её с *наименьшим* числом штрафных ходов. Штрафным назовём такой ход, при котором разрешается перевернуть любую фишку на следующую грань (или на предыдущую, или на единицу), не затрагивая соседних фишек.

С такими послаблениями всякая задача будет разрешима («правильные» задачи решаются без штрафных очков, остальные – с тем или иным количеством штрафных очков), но вот насколько легко будет найти *лучшее* решение?

2. Ни одна фишка на поле не выставляется, а конечная позиция является начальной. *Цель игры*: перевернуть все фишки единицей вверх. Ход выполняется так же, как в игре *ShortGame*. Различие только в том, что теперь у фишек не две стороны, а четыре. Усложнит ли это решение задачи?

3. Придумайте другие варианты игры *Logos*. Самый простой путь – искать их на стыке с другими играми («гибридизация» игр).

Факультатив 5. Цветные линии, или Критическая масса мыслей

Не каждое лыко в строку.

Русская пословица

Прежде чем мы приступим к разработке собственной программы, давайте познакомимся с мировым опытом в области собирания шариков.

Мир *Цветных линий*

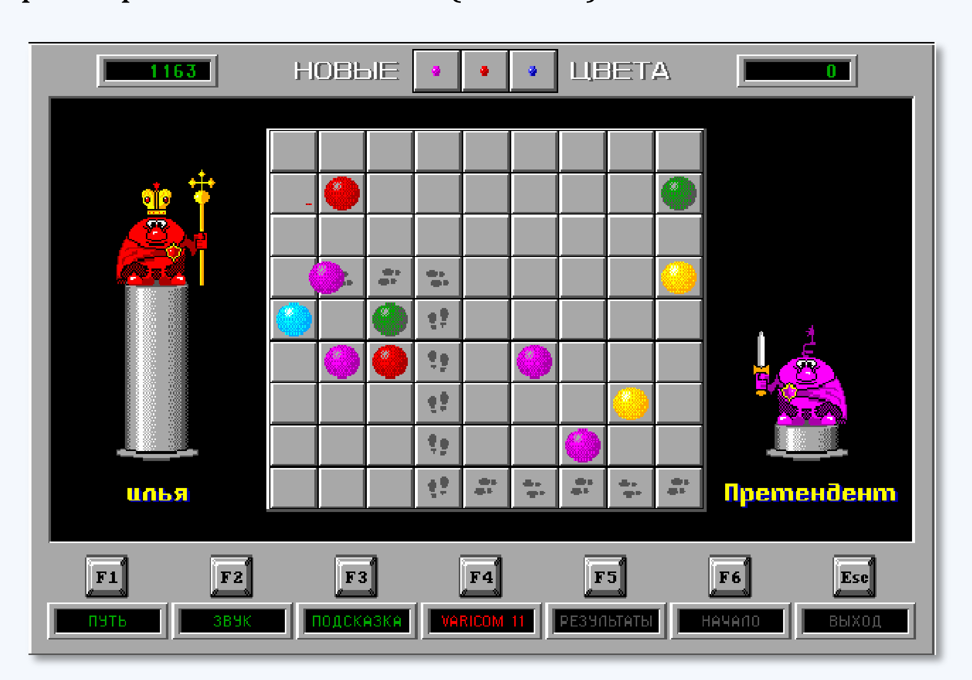
Цветные линии (или *Color Lines*, на аглицкий манер) - одна из самых известных компьютерных игрушек (Рис. 5.1), была создана в недрах фирмы *Геймос* усилиями Олега Дёмина ещё в те времена, когда компьютерами управляла операционная система *MS DOS*. С тех пор она породила, пожалуй, не меньше вариантов, чем знаменитый тетрис.



Рис. 5.1. Мировая вещь!

И это не удивительно: правила игры очень просты и легко поддаются программированию, а сама игра настолько интересна, что от неё сначала трудно оторваться, а потом хочется вновь к ней возвращаться. Наиграться в неё невозможно!

В классическом варианте, представленном здесь программами *Color Lines* (версия для *DOS*, автор Олег Дёмин, 1992 год) и *LinesWin* той же фирмы *Геймос*, но уже для *Windows* (авторы Геннадий Денисов, Игорь Ивкин, Константин Миронович, 1995 год) игра ведётся на квадратном поле размерами 9 на 9 клеток (Рис. 5.2).



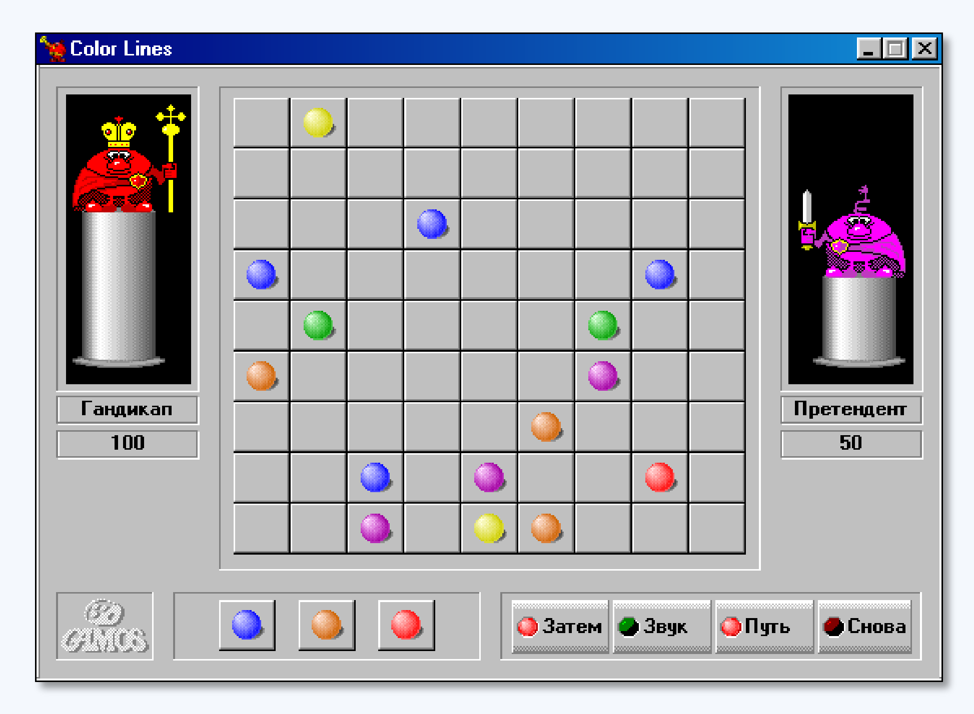


Рис. 5.2. «Исходники»!

При старте игры на поле появляются 5 разноцветных шариков (всего имеется 7 разных окрасок). Игрок на каждом ходу перемещает любой шарик в другую клетку, если он сможет добраться до неё ходом шахматной ладьи. Когда удастся выстроить непрерывную цепь шариков одного цвета длиной не менее пяти штук по горизонтали, вертикали или

диагонали, она (само)уничтожается, освобождая поле, а игрок получает по очку за каждый шарик, плюс возможность сделать ещё один ход. Если на поле одновременно возникает несколько цепочек, то все они будут «стёрты с лица земли».

Если в результате хода выстроить такой цепочки не удалось, то на поле в произвольных местах возникают 3 новых шарика. Вы можете заранее узнать цвет следующей порции шариков, если включите режим подсказки, но тогда призовых очков получите меньше.

Игра заканчивается, когда на поле не останется места для вывода очередных шариков.

Цель игры: набрать как можно больше очков и попасть в список рекордсменов. Легко понять, что никакого логического выигрыша в игре нет, так как отсутствуют выигрышные ситуации (если не учитывать почти невероятный случай, когда игроку удастся уничтожить *все* шарики на поле!). Игрок в любом случае проигрывает, разница только в том, что кто-то сможет продержаться дольше и набрать больше очков. Подобную ситуацию мы наблюдаем и в тетрисе, где также победить нельзя. Но вы можете разбить игру (свой вариант!) на несколько уровней с переходом на следующий после набора определённого количества очков.

Если не учитывать различий в качестве графики, то обе версии игры совершенно одинаковы. Существуют только два технических отличия – шарики по-разному перемещаются по клеткам, и в *ДОС*-версии шарик оставляет за собой следы, что довольно забавно.

Недалеко ушла от оригинала игра *Цветные линии* (Антон Мелёхин, версия 1.1, 1997 год) (Рис. 5.3).



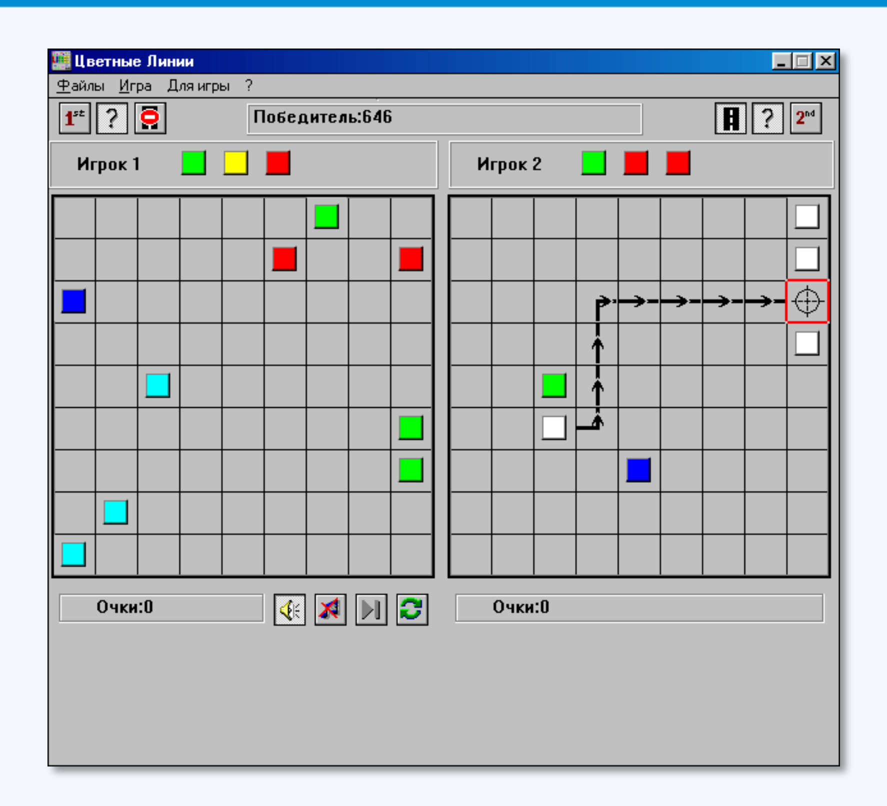


Рис. 5.3. Цветные линии Антона Мелёхина

Написана она ещё на первой версии нашего родного *Delphi*. Игра ведётся по классическим правилам, только в начале игры появляются 3 шарика, что явно недостаточно, так как практически невозможно сделать ни одного осмысленного хода. За каждую уничтоженную цепочку начисляется удвоенное количество очков.

Предусмотрена возможность игры вдвоём, но каждый игрок ведёт совершенно самостоятельную партию (один управляет мышкой, второй - клавиатурой), так что никакого резона в таком усовершенствовании игры нет. Чтобы внести элемент соревновательности, следовало бы играть параллельно, но тогда один игрок может просто повторять ходы другого. Правильнее было бы соревноваться с компьютером, то есть второго игрока должна заменить программа.Для этого потребуется наделить её каким бы то ни было интеллектом, чтобы она выбирала хорошие ходы (или лучшие!) из нескольких возможных.

По ходу игры вы можете слушать музыку. Сейчас это стало модным атрибутом игровых программ, хотя совершенно необязательно интеллектуальные игры портить музыкальным сопровождением. Тем более что каждый меломан может запустить, например, в *Универсальном проигрывателе*, любимую музыку в фоновом режиме.

К достоинствам программы можно отнести динамическое выделение предполагаемого пути шарика стрелками. Очень удобно и наглядно!

Две следующие программы - *Lines'98* (Рис. 5.4)

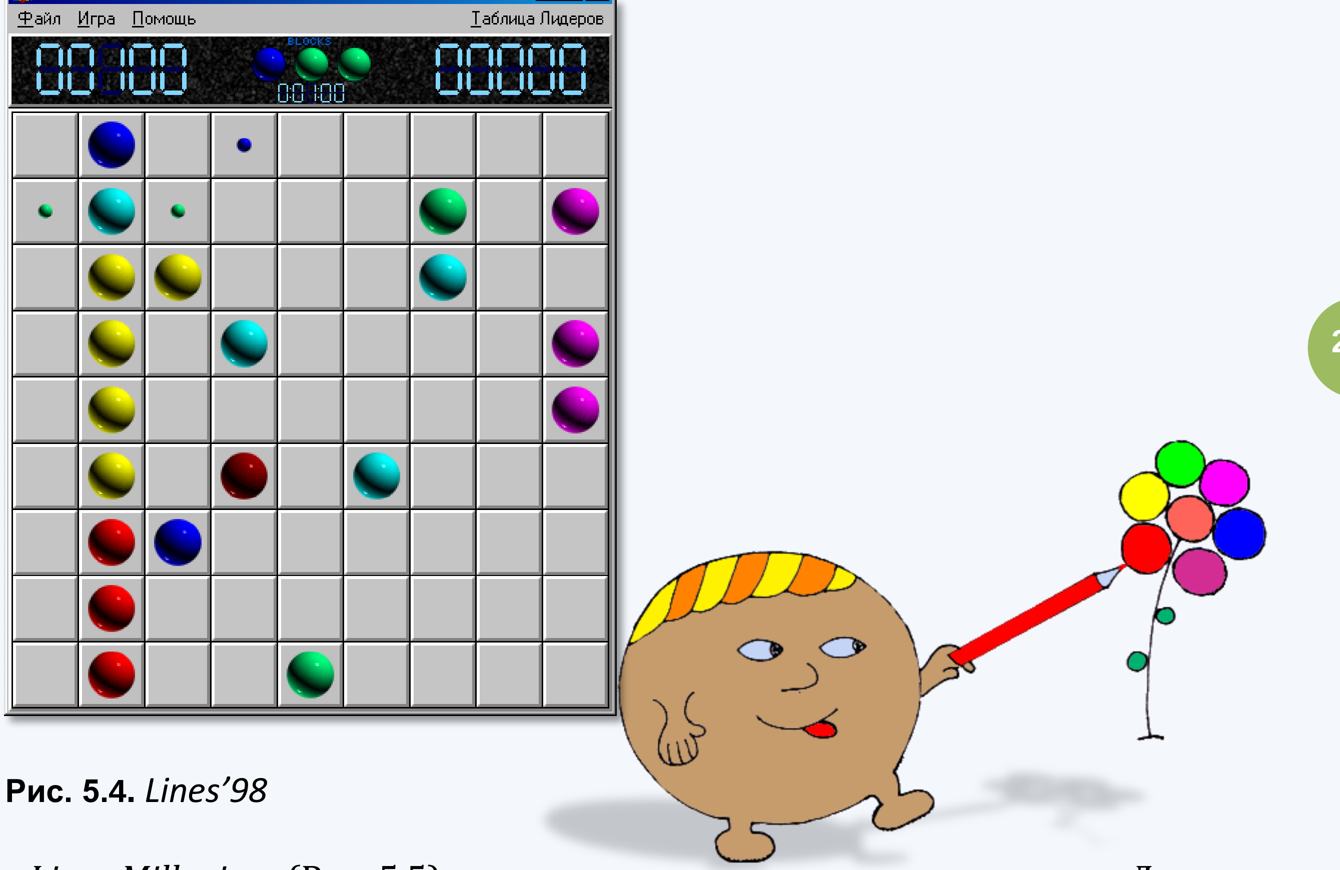
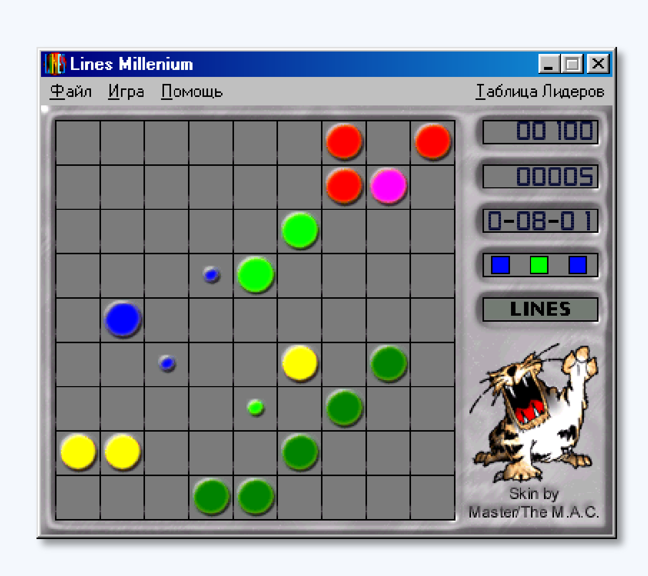


Рис. 5.4. *Lines'98*

и *Lines Millenium* (Рис. 5.5) – принадлежат «перу» одного автора – Дмитрия Кивилёва. Это всё та же классическая игра, но в новом обличии, точнее – во множестве обличий, так как вы можете изменить её внешний вид, выбирая один из нескольких возможных *скинов* (*skins*) – модная нынче технология «пластической операции» интерфейса.



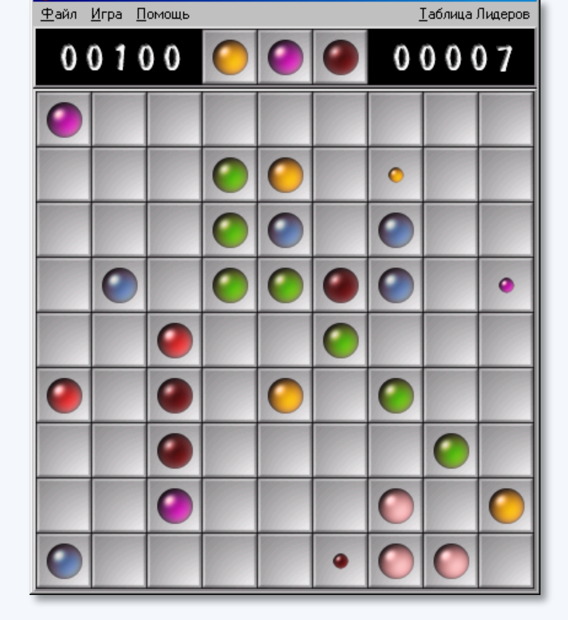


Рис. 5.5. *Lines Millenium*

Кроме стандартной игры, в которой требуется уничтожать линии (*Lines*), имеется ещё две её разновидности - *Squares* и *Blocks*. В первой нужно собирать прямоугольники (не менее четырёх шариков), во второй - связные области (не менее семи шариков).

Другие нововведения: прогрессивное начисление очков в зависимости от длины цепочки, выбор и настройка звуковых эффектов для основных событий, сохранение и загрузка игры, откат на один ход, изменение количества вновь появляющихся шариков (3-5), следующие шарики могут отображаться не только на табло, но и на поле (шарики меньшего размера).

В целом игра оставляет весьма приятное впечатление.

Несомненно, порадует вас обилием скинов и игра *Cool Lines* (автор Настя Рассохина, 2000 год) (Рис. 5.6), тем более что их легко изготовить самостоятельно.





Рис. 5.6. *Cool Lines*

В остальном эта игра лишь незначительно отличается от оригинала:

- можно уничтожать не только линии, но и квадраты из четырёх «шариков»;

- отсчитывается время игры;

- количество вновь появляющихся фигурок непостоянно (5-9) в ходе одной игры;

- имеется 3 уровня сложности, которые и определяют набор фигурок.

В игре *Picture Lines 1.0* (автор Вячеслав Витер, 2000 год) тоже поддерживаются скины (Рис. 5.7).



Рис. 5.7. *Picture Lines 1.0*

Игра проходит на поле стандартных размеров, но имеется возможность выбирать количество вновь появляющихся шариков (3-5) и их цвета (5-9). Уничтожаются не только прямые линии, но и прямоугольники, прямые и косые кресты и блоки (связные области). Выводится время игры, процент занятой шариками территории, путь шарика. Текущую игру можно сохранить на диске, а затем вновь загрузить.

Из новинок можно отметить появляющуюся по мере уничтожения шариков картинку, что, впрочем, ничуть не улучшает игру, так как б***о***льшая часть поля всё равно занята шариками. Было бы лучше вынести проявляющуюся картинку на отдельную панель или форму.

Всё бы ничего, но назойливая реклама над игровым полем не способствует интересу к этой игре, тем более что ничего принципиально нового в ней нет.

В игре *Цветные линии 1.01* (Cheensoft, 1997 год), можно отметить только указатель заполнения поля шариками на компоненте *ProgressBar* (Рис. 5.8).



Рис. 5.8. *Цветные линии 1.01*

В игре *Alhademic Lines 1.20* (Alhademic Group, 2000 год) (Рис. 5.9) вы найдёте красивую графику (выбор из нескольких скинов и наборов фишек), приятную музыку и удобный интерфейс.

Собирать можно не только линии, но также прямоугольники и блоки. Имеется возврат последнего хода, три уровня сложности (6-8 цветов шариков), динамический указатель пути, подсказки на табло и на поле). Если вы не ищете экзотики, то эта игра для вас.



Рис. 5.9. *Alhademic Lines*

Игры *Тёмные линии* (*DarkLines 2000 Elite 2.1*) (Рис. 5.10) и *DarkLines 3: ERA* (2001 год) (Рис. 5.11) имеют игровое поле 10 на 10 клеток, позволяют изменять скины и вид фишек.

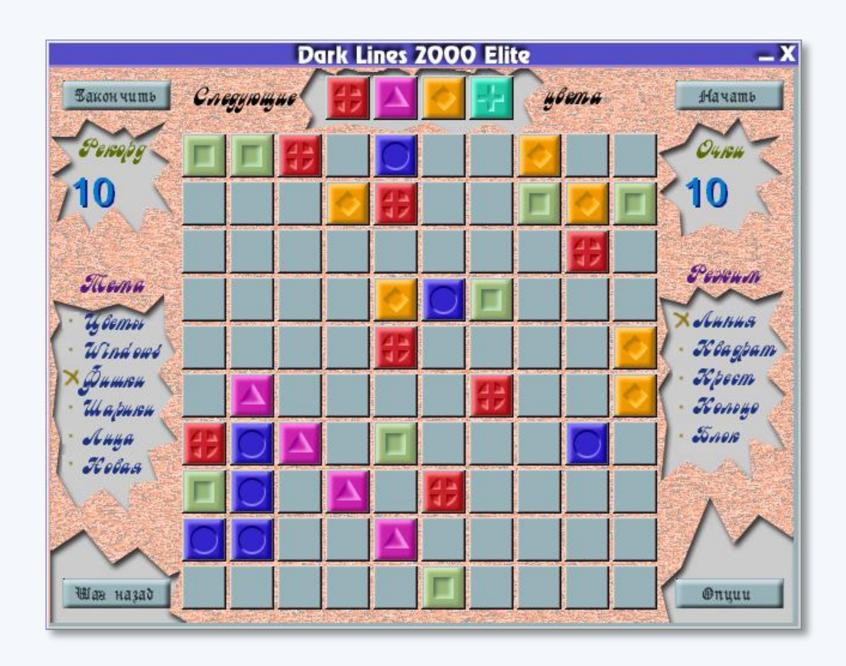


Рис. 5.10. *Тёмные линии*

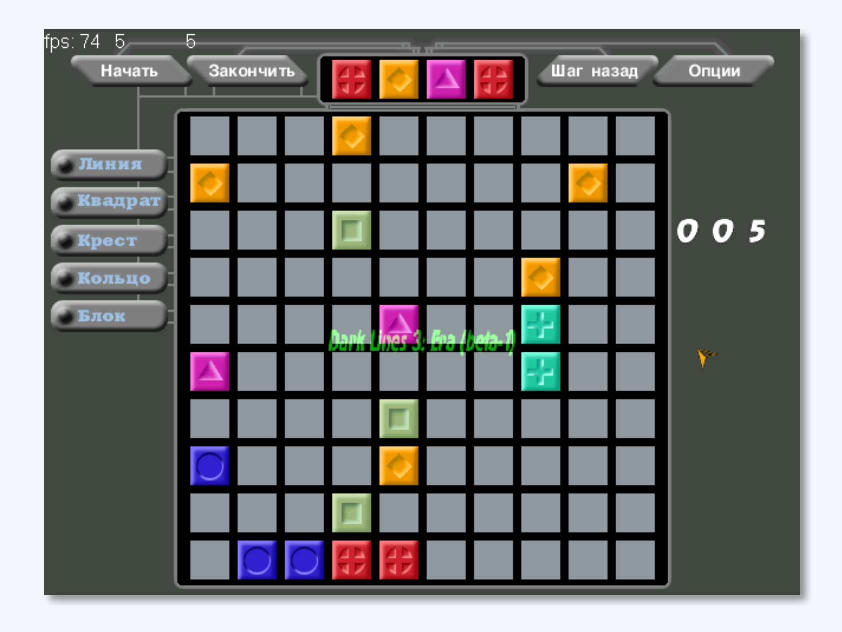


Рис. 5.11. *Тёмные DarkLines 3: ERA*

Из дополнительных возможностей можно назвать: откат на 1 ход назад, несколько уровней сложности игры (в усложнённом варианте часть клеток поля на время блокируется – с них и на них нельзя ходить), появляется не 3, а 4 новых фишки, составлять можно не только линии, но и прямоугольники, кольца, кресты, углы или блоки.

5 Star Free Lines 2.1 (2001 год) (Рис. 5.12).



Рис. 5.12. *5 Star Free Lines 2.1*

Игра ведётся по классическим правилам, но имеется особый шарик – *джокер*, который может заменить шарик любого цвета. Очки начисляются прогрессивно - в зависимости от длины цепочки.

В игре *Color Lines 99* (Рис. 5.13) Максима Иванова вас порадуют: несколько видов оформления поля, интересная заставка и – классические правила, осложнённые хитрым начислением набранных очков.



Рис. 5.13. *Color Lines 99*

В игре *Magic Lines* (версия 3.6, 2006 год, А.Сергеев, П.Тарасов) (Рис. 5.14) следует в первую очередь отметить интересную графику: вращающиеся объёмные шарики, освещение которых меняется в зависимости от положения курсора мыши. От классического варианта эта игра отличается только тем, что сразу появляются только три шарика, а следующие показываются не только на табло, но и на поле.

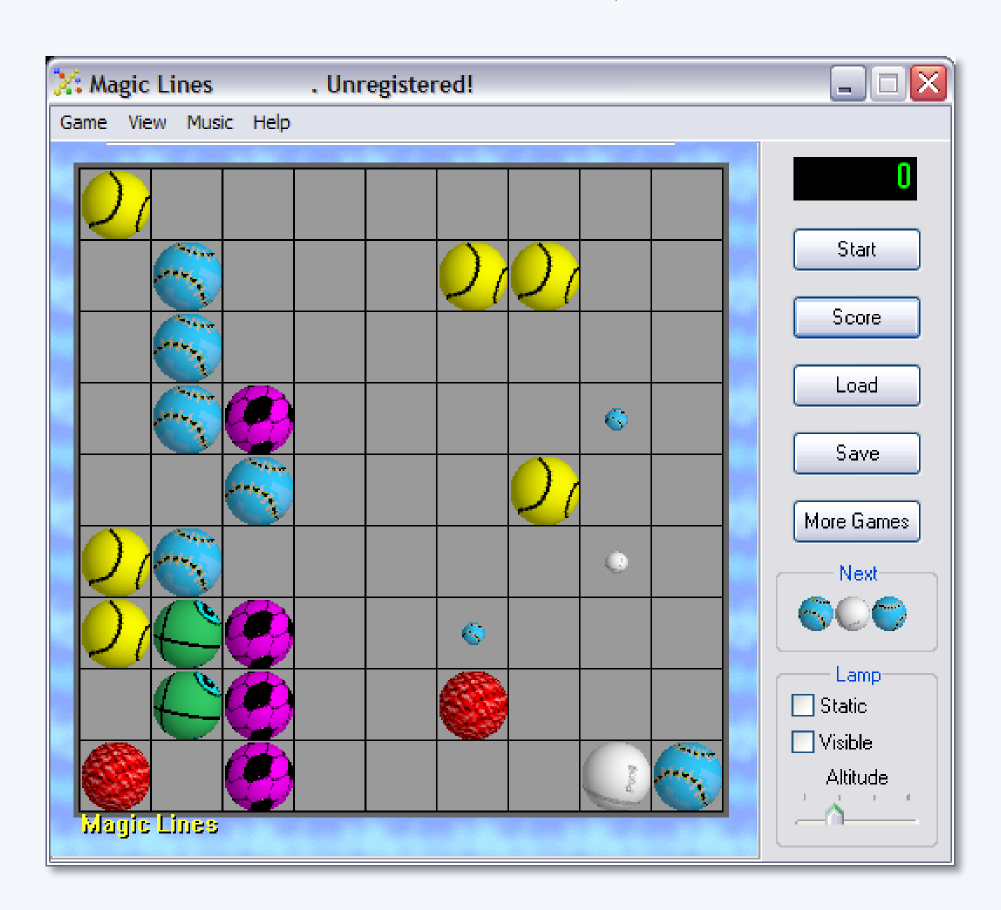


Рис. 5.14. *Magic Lines*

3D Lines 1.2 (автор Александр Шелемехов, 2001 год) - настоящая *трёхмерная* игра (Рис. 5.15).

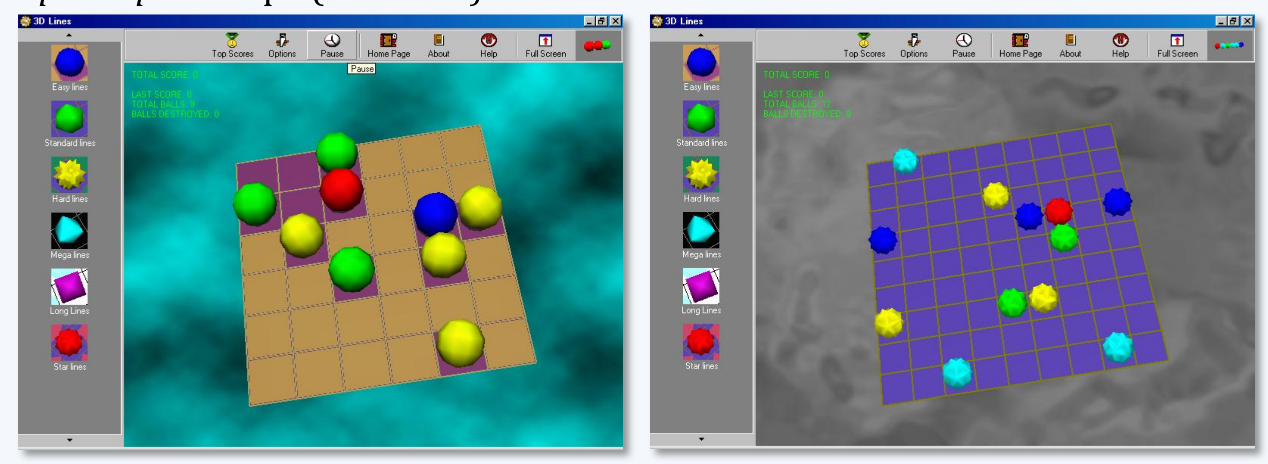


Рис. 5.15. *3D Lines 1.2*

В *Honey Lines* (Nikita, 1995 год) игра происходит на *гексагональном* поле (Рис. 5.16), что не только радует глаз, но и облегчает задачу игрока, ведь теперь линии можно выстраивать не по четырём, а по *шести* направлениям. Более того, разрешается не только перемещать фишки на свободные клетки, но и менять местами две соседние и даже пропустить очередной ход, хотя пользы от этого не много.



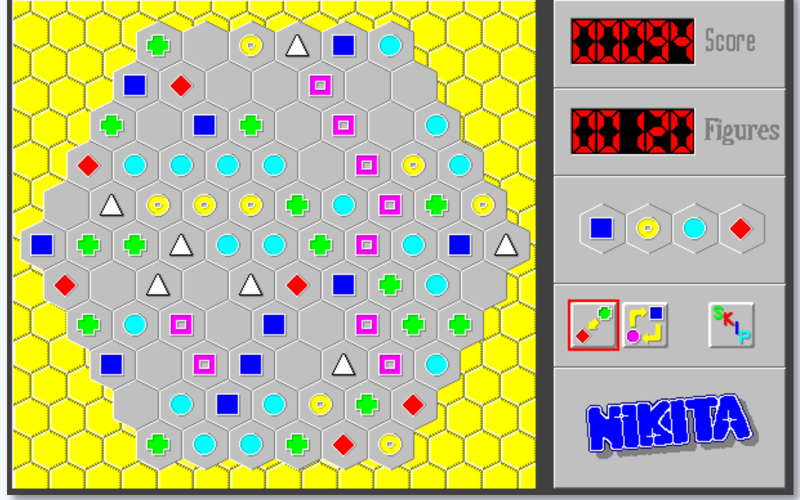


Рис. 5.16. *Honey Lines*

Очень любопытный вариант игры, правда, слегка омрачённый плохо подобранной музыкой и звуковыми эффектами. Ну, и ложка дёгтя в бочку мёда: так как задача игрока - не наполнение сотов, а их опустошение, то, скорее, он не пчёлка, а трутень. Но тут как в сказке: добро всегда побеждает зло, а соты всё рано будут наполнены мёдом, сколько бы вы его ни уничтожали!

Нить Ариадны, или Как выбраться из лабиринта

На первый взгляд, в игре *Цветные линии* нет никакого *лабиринта*. А всё-таки он есть!

Вот вы выбрали шарик и указали клетку, на которую он должен перейти. Тут и возникают вопросы: *возможен* ли такой переход и, если возможен, то *как* найти *самый короткий* путь из одной клетки в другую.



В данной игре это необязательное условие, но и гонять шарик по всему полю тоже негуманно; во многих других играх придётся искать именно *кратчайший* путь, так что лучше подготовиться к этому заблаговременно.

Поскольку целый класс задач посвящён выбору правильного пути, то мы рассмотрим эту проблему более подробно. Оставим на время нашу программу и сыграем этюд на тему *лабиринта*.

Итак, откройте новый проект и установите на форме сетку *dgPole* размером 12 х 12 клеток. Для хранения информации о лабиринте заведите массив *masPole:*

```
masPole: array[0..11, 0..11] of integer;
```

Как известно, в лабиринте есть *проходимые* и *непроходимые* клетки (в реальных лабиринтах, разумеется, клеток нет, а вот в головоломных – чаще всего имеются). Обозначим первые константой *GO*, вторые – *STOP*. Значения им можно присвоить любые *отрицательные*, чтобы уметь отличать их от номеров, которые мы будем записывать в клетки:

```
const
  GO= -1;
  STOP= -2;
```

Прежде всего, научимся из пустой сетки делать лабиринт. Вообще-то над хорошим лабиринтом нужно изрядно поработать (вспомните Дедала, знаменитого строителя первого лабиринта на острове Крит), но для нас качество лабиринта никакого значения не имеет, поэтому мы будем просто закрашивать клетки *белым* (проходимые) и *чёрным* (непроходимые) цветом.

Сложность лабиринта определяется, в том числе, количеством непроходимых клеток на поле, поэтому вы можете создавать лабиринты любой сложности, изменяя значение переменной *Dif*. А дальше всё просто:

случайным образом мы расставляем чёрные и белые клетки на поле – и лабиринт готов:

```
//СОЗДАТЬ ЛАБИРИНТ
procedure TForm1.CreateLab;
var
  i, j: integer;
begin

  for j:= 0 to 11 do
    for i:= 0 to 11 do
      if Random(101) >= Dif then
        masPole[i,j]:= GO
      else
        masPole[i,j]:= STOP;
  dgPole.Invalidate;
end;
```


Он будет появляться при старте программы:

```
//СТАРТ ПРОГРАММЫ
procedure TForm1.FormCreate(Sender: TObject);
begin
  Randomize;
  CreateLab;
end;
```

А если он вам не понравится, вы в любой момент можете его заменить другим, нажав кнопку:

```
//НОВЫЙ ЛАБИРИНТ
procedure TForm1.Button1Click(Sender: TObject);
begin
  CreateLab;
end;
```

Для изменения «сложности» лабиринта (по сути, числа препятствий), установим на поле *скроллер sbDif*:

```
//УСТАНОВИТЬ СЛОЖНОСТЬ ЛАБИРИНТА 0..100%
procedure TForm1.sbDifChange(Sender: TObject);
begin
  dif:= sbDif.Position;
  lblDif.Caption:= inttostr(Dif);
  CreateLab;
end;
```

Теперь нужно выделить в сетке *начальную* и *конечную* клетки. Так как это только набросок для будущих программ, то мы не будем тратить много времени на этот процесс. Поступим просто: нажав *левую* кнопку мыши, мы

зададим *начальную* клетку, *правую* – *конечную*. Под координаты этих клеток отведём *переменные*:

```
BeginCell, EndCell: TPoint;
```

А числа в этих клетках обозначим *константами*

```
  BEG_CELL= 0;
  END_CELL= -3;
```


END_CELL может быть любым *отрицательным* (или большим положительным) числом, а вот *BEG_CELL* непременно должна быть равна *нулю*, так как именно с этой клетки и начнётся поиск короткого пути. Подробности смотрите в описании алгоритма поиска.

```
//ВЫДЕЛИТЬ КЛЕТКИ
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
begin
  //координаты мыши:
  dgPole.MouseToCell(x,y,ACol,ARow);

  //если нажата левая кнопка мыши, то отмечаем начальную клетк
  if ssLeft in shift then begin

     //проходимая клетка:
     if ssCtrl in shift then
       masPole[ACol,ARow]:= GO
     else if ssAlt in shift then
       masPole[ACol,ARow]:= STOP
     //непроходимая клетка:
     else begin
       BeginCell:= Point(ACol,ARow);
       masPole[ACol,ARow]:= BEG_CELL
     end

  end
  //отметить конечную точку:
  else begin
    EndCell:= Point(ACol,ARow);
    masPole[ACol,ARow]:= END_CELL;
  end;

  dgPole.Invalidate
end;
```


Не всякий лабиринт придётся вам по вкусу, поэтому у вас должна быть возможность самостоятельно конструировать лабиринт. Сделать это очень просто: щёлкая *левой* кнопкой мыши, удерживайте клавишу *CTRL*,

чтобы поставить *проходимую* клетку, или *ALT*, чтобы запретить проход через данную клетку.

Для запуска процедуры поиска пути мы установим на форме ещё одну кнопку.

Вот так примерно может выглядеть интерфейс этой программы (Рис. 5.17, сдева).





Рис. 5.17. Интерфейс программы Путь в лабиринте найден!

Итак, отмечаем клетки и нажимаем кнопку *Найти путь*. Если путь существует, он будет показан на экране (Рис. 5.17, справа), в противном случае появится только соответствующая надпись:

```
//ИСКАТЬ ПУТЬ
procedure TForm1.Button2Click(Sender: TObject);
begin
  if FindPath then
    ShowPath
  else
    showmessage('Пути нет')
end;
```

Есть несколько *алгоритмов* поиска пути в лабиринте, мы воспользуемся наиболее эффективным из них, который называется *поиском в ширину* (в

применении к поиску пути в лабиринте он называется также *методом волновой трассировки*, или *волновым алгоритмом*).

Заведём массив *CoordList*, в который будем записывать координаты тех клеток, которые следует рассмотреть. Сначала мы помещаем в него координаты начальной точки *BeginCell*. Переменная *ptrRead* всегда указывает на очередную анализируемую клетку в этом списке. Так как пока в списке только одна клетка, *ptrRead* указывает на неё. На каждом шаге мы извлекаем из списка координаты очередной клетки и проверяем всех её соседей. В лабиринте разрешается двигаться только влево-вправо и вниз-вверх и только по *проходимым* клеткам (в них записано число *GO* или *END_CELL* – для *конечной* клетки). Найдя подходящую клетку, мы дописываем её координаты в конец списка *CoordList*. Индекс для записи хранится в переменной *ptrWrite*. В начале поиска мы присваиваем ей значение *2*, так как это первый свободный индекс в массиве *CoordList*. После того как записаны координаты новой клетки, указатель *ptrWrite* смещается на следующий индекс. Кроме того, в саму клетку мы записываем число, на единицу большее, чем в текущей клетке. Оно, как легко понять, показывает *минимальное* число шагов, необходимое для перехода в эту клетку из начальной. Именно поэтому в начальную клетку мы и записали *0*. Во все соседние проходимые клетки будет записана *единица*. Во все проходимые (и ещё не отмеченные!) клетки, соседствующие с ними, - *двойка*, и так далее.

Этот процесс хорошо виден на следующих двух картинках (Рис. 5.18).

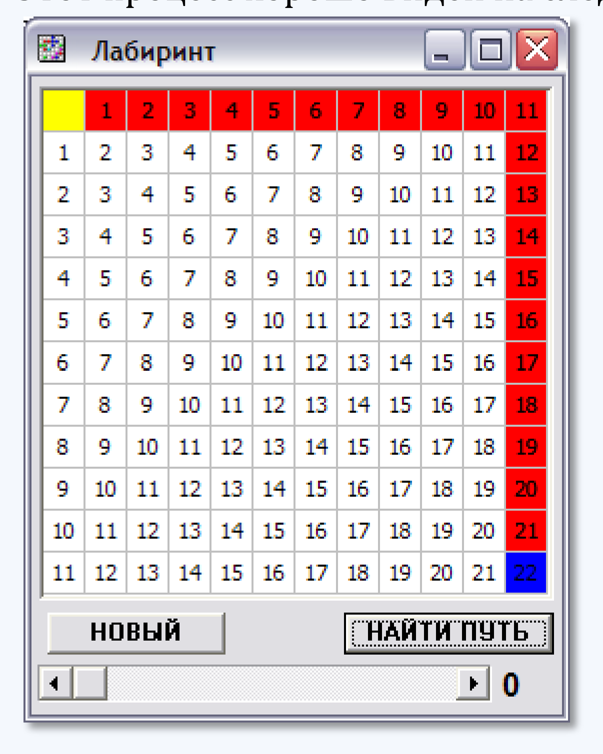




Рис. 5.18. Погнали волну по лабиринту!

Внимательно проследите, как программа нумерует клетки! Это полезное занятие приведёт вас к ответу, почему алгоритм – весьма образно! – называется *волновым*.

Поиск заканчивается либо при достижении *конечной* клетки (тогда функция *FindPath* возвращает *True*), либо при исчерпании списка (путь не найден, функция возвращает *False*).

```
//НАЙТИ ПУТЬ МЕЖДУ ДВУМЯ ТОЧКАМИ
function TForm1.FindPath: boolean;
var
  //список координат:
  CoordList: array[1..143] of TPoint;
  //указатели в списке:
  ptrWrite, ptrRead: integer;
  p, q: integer;
  i, j: integer;

  //проверить координаты
  function TestCoord(x,y: integer): boolean;
  begin
    Result:= true;
    if (x<0) or (x>11) or (y<0) or (y> 11)or
     ((masPole[x,y]<> GO) and (masPole[x,y]<> END_CELL)) then
      Result:= false;
  end;

begin
  //если BeginCell = EndCell, то начальная клетка совпадает с конечной,
  //и путь искать не нужно!
  //заносим в список координаты начальной клетки:
  CoordList[1]:= BeginCell;
  //устанавливаем указатель для считывания координат на начало списка:
  ptrRead:= 1;
  //устанавливаем указатель для записи новых координат на следующий ин-
декс:
  ptrWrite:= 2;
  //в начальной клетке в массиве masPole находится BEG_CELL= 0

  //двигаемся от начала списка к его концу, пока он не кончится:
  while ptrRead < ptrWrite do begin
    //координаты текущей клетки:
    p:= CoordList[ptrRead].x; q:= CoordList[ptrRead].y;
    //проверяем соседние с ней клетки:
    for i:= p - 1 to p + 1 do
      for j:= q - 1 to q + 1 do
        //если нашли соседнюю проходимую клетку,
        if ((i=p) or (j=q)) and TestCoord(i,j) then
        begin
          //то записываем в неё число, на единицу большее,
          //чем в текущей клетке:
          masPole[i,j]:= masPole[p,q] + 1;
          //если дошли до конечной клетки,
          if (i= EndCell.x) and (j= EndCell.y) then begin
            //то путь найден:
```

```
            Result:= True;
            exit;
          end
          else begin
            //записываем координаты соседней клетки в конец списка:
            CoordList[ptrWrite]:= Point(i,j);
            //перемещаем указатель:
            inc(ptrWrite);
            dgPole.Invalidate;
            //showmessage(inttostr(masPole[i,j]) + ' x='+inttostr(i)+ '
y='+inttostr(j));
          end;
        end;
      //переходим к следующей клетке в списке:
      inc(ptrRead);
  end;
  //путь не найден:
  Result:= False;
end;
```


За работой алгоритма удобно наблюдать в *пошаговом* режиме. Раскомментируйте эту строку, и вы сможете отслеживать все изменения в сетке:

```
showmessage(inttostr(masPole[i,j]) + ' x='+inttostr(i)+ ' y='+inttostr(j));
```

Легко заметить, что этот же алгоритм можно использовать и для *закрашивания* области поля в какой-либо цвет. Для этого достаточно в качестве начальной клетки (или точки) выбрать любую клетку внутри области, а координаты конечной точки задать *отрицательными*. В этом случае путь найти невозможно, поэтому вся область будет заполнена числами или закрашена выбранным цветом.

Все изменения в массиве *masPole* нужно отразить на экране:

```
//ОТРИСОВАТЬ СЕТКУ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow: Inte-
ger;
  Rect: TRect; State: TGridDrawState);
var
  s: string;
begin
  //закрасить клетку своим цветом:
  case masPole[ACol, ARow] of
    GO: dgPole.Canvas.Brush.Color:= clWhite;
    STOP: dgPole.Canvas.Brush.Color:= clBlack;
    BEG_CELL: dgPole.Canvas.Brush.Color:= clYellow;
    END_CELL: dgPole.Canvas.Brush.Color:= clBlue;
    //номер хода:
    else begin
       with Rect, dgPole.Canvas do begin
```

```
        Brush.Style:= bsClear;
        s:= inttostr(masPole[ACol, ARow]);
        textrect(Rect, left+(right-left-textwidth(s)) div 2,
             top+(bottom-top-textheight(s)) div 2, s);
      end;
    end;
  end;
  dgPole.Canvas.FillRect(Rect);
end;
```

Теперь мы знаем, существует ли путь между двумя точками или нет, но как его найти? А очень просто! Нужно двигаться от *конечной* точки к *начальной* по тем числам, которые мы записали в клетки. Очевидно, что в конечной клетке находится самое большое число, равное длине всего пути. Попасть в неё можно только из клетки с числом на единицу меньше. И так далее, пока мы не достигнем точки с нулём, которая служит *начальной*. По пути выделяем на экране все клетки пути *красным* цветом.


```
//ПОКАЗАТЬ  ПУТЬ
procedure TForm1.ShowPath;
var
  n, LenPath: integer;
  i, j, p, q: integer;
  path: array[0..144] of TPoint;
  Rect: TRECT;
  s: string;

  //проверить координаты:
  function TestCoord(x,y: integer): boolean;
  begin
    Result:= true;
    if (x<0) or (x>11) or (y<0) or (y> 11)or (masPole[x,y]<> n-1) then
      Result:= false;
  end;

begin
  //длина пути равна числу в конечной клетке:
  LenPath:= masPole[EndCell.x, EndCell.y];
  n:= LenPath;
  //конечная клетка пути:
  path[n]:= EndCell;
  //двигаемся от неё к начальной клетке:
  repeat
    //найти соседнюю клетку с числом n-1:
    p:= path[n].x;  q:= path[n].y;
    //проверяем соседние клетки:
    for i:= p - 1 to p + 1 do
      for j:= q - 1 to q + 1 do
        //нашли подходящую клетку:
        if ((i=p) or (j=q)) and TestCoord(i,j) then
        begin
          //записываем её координаты:
          path[n-1]:= Point(i,j);
          break;
```

```
        end;
    //ищем клетку с предыдущим номером:
    dec (n);
  until n<0;
  //показать путь в сетке:
  for i:= 1 to LenPath-1 do begin
    {ListBox1.Items.Add(inttostr(i)+ ' ' + inttostr(path[i].x)+ ' '+
                            inttostr(path[i].y));}
    Rect:= dgPole.CellRect(path[i].x,path[i].y);
    //выделить красным цветом:
    dgPole.Canvas.Brush.Color:= clRed;
    dgPole.Canvas.FillRect(Rect);
     with Rect, dgPole.Canvas do begin
        s:= inttostr(i);
        textrect(Rect, left+(right-left-textwidth(s)) div 2,
             top+(bottom-top-textheight(s)) div 2, s);
     end;
  end;
end;
```


Действуя таким образом, мы найдём только *один* из возможных (и самых коротких!) путей. Нам пока большего и не надо, но, если вам придётся проверять *единственность* решения какой-либо задачи, то имейте это в виду!



Исходный код программы находится в папке **lab**.

Большая стройка, или «Линейное» программирование

Подковавшись практически и теоретически, мы вполне созрели до собственноручного построения программы!

Не забудьте установить компонент с нашей кнопкой! Впрочем, вы легко можете избавиться от нововведений, если замените кнопки *TEasyButton* стандартными. Для этого удалите новые кнопки, установите те кнопки, которые вам нравятся, и настройте их свойства и обработчики событий.

Начнём с *формы*. Она будет простой, прямоугольной, но без строки заголовка (*BorderStyle=bsNone*), которая в данном случае только портила бы вид окна приложения. Заголовок мы заменим картинкой в компоненте *Image1* (картинки находятся в папке *Res/button*). Перемещать окно приложения придётся также за картинку:

```
  private
```

```
    { Private declarations }
    rgn: HRGN;
    procedure WMNCHitTest( var Msg: TWMNCHitTest); message
WM_NCHITTEST;
```

```
//ПЕРЕМЕЩЕНИЕ ОКНА
procedure TForm1.WMNCHitTest( var Msg: TWMNCHitTest);
begin
  Inherited;
  WITH Msg DO
    WITH ScreenToClient(Point(XPos,YPos)) DO
      if PtInRegion(rgn,X, Y) then
         Result:= HTCAPTION
      else Result:= HTNOWHERE
end;
```


Когда мы нажимаем кнопку мыши на картинке, эта процедура указывает приложению, что мышка будто бы находится на заголовке окна. Но сначала мы должны сформировать *область* по форме картинки *Image1*. Это удобно сделать при запуске приложения:

```
/////////////////////////////////////////////////////////////
//////////////////////// ФОРМА ///////////////////////////
/////////////////////////////////////////////////////////////

//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
  //цвет фона:
  fon:= rgb(120,144,120);
  Color:= fon;
  Panel1.Color:= fon;
  Panel2.Color:= fon;
  dgNextBalls.Color:= fon;
  //создать поле:
  Pole:= tPole.Create;
  //очки за игру:
  Score:= tScore.Create;

  //сформировать "заголовок окна":
  with image1 do
    rgn:= CreateRectRgn(Left,Top,Left+Width,top+ Height);

  //размеры поля по умолчанию:
  Pole.Width:= 9;
  Pole.Height:= 9;
  //шарики:
  NumBalls.nSrazu:= 5;
  NumBalls.nPotom:= 3;
  NumBalls.nColor:= 7;

  TypeFig:=tfLines;

  //размеры клетки сетки:
  wCell:= dgPole.DefaultColWidth;
  hCell:= dgPole.DefaultRowHeight;
end;
```

Здесь же мы окрашиваем окно приложения и компоненты в какой-нибудь забавный цвет и производим настройки игры, речь о которых пойдёт ниже. Непосредственная подготовка к новой игре будет происходить при активации формы - так мы сможем уже после старта программы приступить к игре, не нажимая никаких кнопок:

```
//ПОДГОТОВИТЬСЯ К ИГРЕ
procedure TForm1.FormActivate(Sender: TObject);
begin
  Prepare;
  delay(1000);
  Pole.StartGame;
end;
```


Конечно, при этом игра стартует со свойствами по умолчанию:

- размеры поля 9 х 9 клеток;

- сразу появятся 5 шариков, а дальше каждый раз будут появляться 3 шарика;

- всего в игре используются шарики 7 разных цветов;

- собирать из шариков нужно линии.

Все эти настройки вы можете видеть в процедуре *FormCreate*.

Чтобы края формы получились более красивыми установите свойство формы *Ctrl3D* в *False*, а в процедуре-обработчике формы *FormPaint* напишите:

```
//ОТРИСОВАТЬ ФОРМУ
procedure TForm1.FormPaint (Sender: TObject);
var
  RECT: TRect;
begin
  //придать объёмный вид краям формы:
  RECT:= ClientRect;
  DrawEdge(Canvas.Handle, Rect, EDGE_RAISED, BF_RECT)
end;
```

Ну и чтобы закончить с формой, забежим немного вперёд и напишем обработчик события *OnFormDestroy*, которое возникает при закрытии приложения:

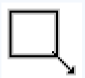
```
//УНИЧТОЖИТЬ ФОРМУ
procedure TForm1.FormDestroy(Sender: TObject);
begin
 Pole.Free;
```

```
 //DeleteObject(rgn);
 Timer1.Enabled:= False;
 Timer2.Enabled:= False;
 Score.Free;
 Tonus.Free;
 hod.Free;
end;
```

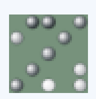
Сущность этих «мероприятий» станет понятна вам немного позже.

Теперь мы украсим форму *кнопками TEasyButton* (или другого типа, если вы передумали). Достаточно шести штук:

 **Name**=btnNewGame **Hint**=Новая игра **ButtonBorde**r= brNone

 **Name**=btnWH **Hint**=Размеры поля

 **Name**=btnOptions **Hint**=Настройки

 **Name**=btnTypeFig **Hint**=Вид собираемых фигур

 **Name**=btnHelp **Hint**=Справка

 **Name**= btnExit **Hint**=Выход из программы

Размеры всех кнопок 40 x 40 пикселей. Картинки для них находятся в папке *Res/button*.

Назначение всех кнопок ясно следует из подсказок.

С некоторыми из них мы можем «разобраться» уже сейчас. Что может быть проще, чем закрыть программу? – Нажимаем кнопку *btnExit* – и всё:

```
//ЗАКРЫТЬ ПРОГРАММУ
procedure TForm1.btnExitClick(Sender: TObject);
begin
  close
end;
```

Как вы помните, при этом возникает событие *OnFormDestroy* и вызывается обработчик этого события.

Немного сложнее вывести справку по программе – кнопка *btnHelp*:

```
//ВЫЗОВ СПРАВКИ
procedure TForm1.btnHelpClick(Sender: TObject);
var s: string;
begin
  s:= extractfilepath(application.exename)+'LinesHelp.htm';
  ShellExecute(handle, 'open','iexplore',pChar(s),
  '', SW_SHOWMAXIMIZED);
end;
```


В самой процедуре, конечно, ничего нового для нас нет, но придётся написать файл справки. Впрочем, в начале главы для этого имеется вся необходимая информация!

Перейдём к следующей кнопке – *btnTypeFig*. В классическом варианте из шариков следует строить *линии*, но мы будем собирать и *блоки*, под которыми мы будем понимать, любые фигуры (в том числе и линии, за исключением «диагональных»), имеющие «площадь» не менее 6 шариков:

```
  //тип собираемых фигурок:
  TypeFig:(tfLines, tfBlocks);
```

Выбирать тип собираемых фигур можно и до, и во время игры, нажав кнопку *btnTypeFig*. Под кнопкой появится меню - дальше следует действовать, как обычно (Рис. 5.19).

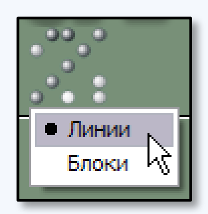


Рис. 5.19. Выбираем тип сборки!

```
//ВЫБРАТЬ ТИП СОБИРАЕМЫХ ФИГУР
procedure TForm1.btnTypeFigClick(Sender: TObject);
var
  p: Tpoint;
begin
  //вывести всплывающее меню:
  p:= ClientToScreen(point(btnTypeFig.Left, btnTypeFig.Top+
btnTypeFig.Height));
  popupmenu2.Popup(p.x, p.y);
  //сбросить рекорд:
  score.Rec:= 0;
end;
```

Установим на форме всплывающее меню *PopupMenu2* и наберём две строки (Рис. 5.20).

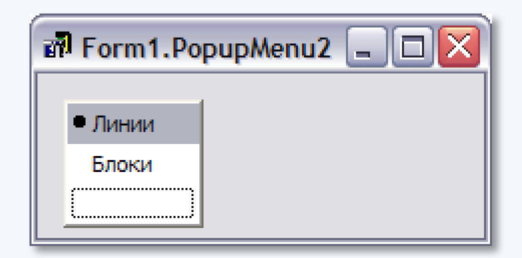


Рис. 5.20. Выбираем тип сборки!

По умолчанию играется *классический* вариант - с *линиями* (*первая* строка, **Name**=*miLines*, **tag**=*1*). После выбора пункта меню программа направляет нас в обработчик события *OnmiLinesClick*:

```
procedure TForm1.miLinesClick(Sender: TObject);
begin
  //показать пункт меню как выбранный:
  (Sender as TMenuItem).Checked:= True;
  if (Sender as TMenuItem).tag=1 then TypeFig:= tfLines
  else TypeFig:= tfBlocks;
  //начать новую игру:
  //NewGame;
end;
```

Если вы хотите, чтобы игра велась без смены фигур, то раскомментируйте строку *//NewGame;*.
Другие настройки игры вызываются кнопкой *btnOptions*. Их довольно много, поэтому мы добавим к проекту ещё одну *форму – frmOptions*, на которой и разместим все элементы управления: *табулятор TabControl1* и *кнопку ГОТОВО* (Рис. 5.21).

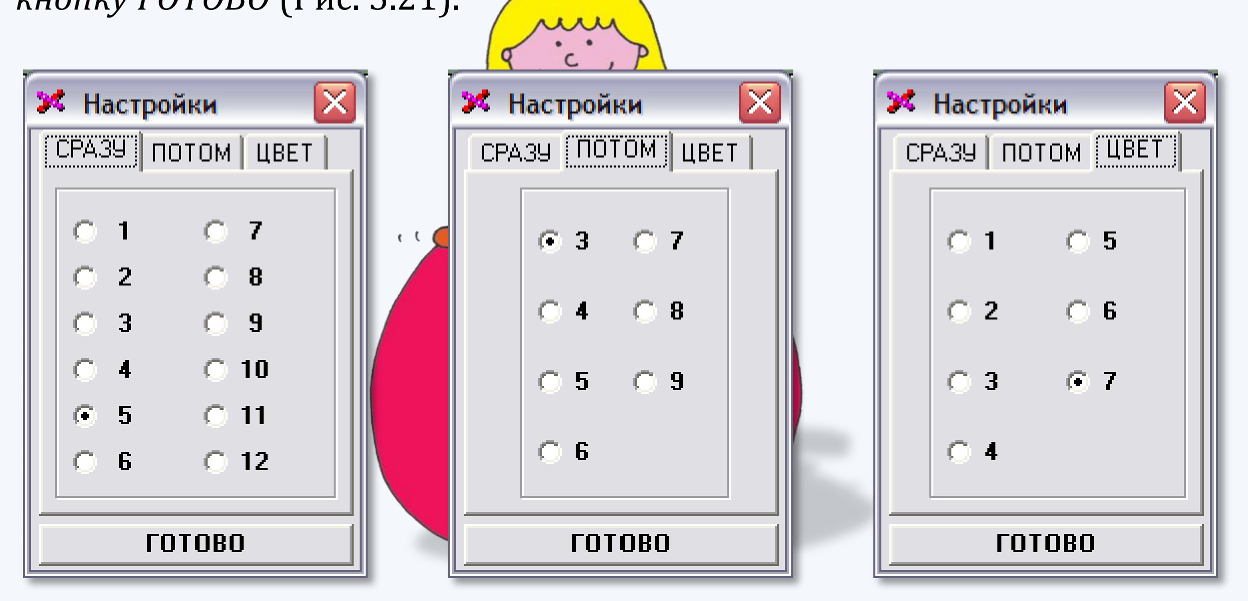


Рис. 5.21. Настройки – по настроению!

Форма будет появляться под кнопкой после нажатия на неё:

```
//НАСТРОЙКА ПАРАМЕТРОВ ИГРЫ
procedure TForm1.btnOptionsClick(Sender: TObject);
begin
  //вывести форму для настройки параметров игры:
  frmOptions.Left:=left+btnWH.Left+btnWH.width;
  frmOptions.Top:= top+btnWH.Top+ btnWH.Height;

  frmOptions.ShowModal;

  //установить новые параметры -

  //число шариков, появляющихся при старте новой игры:
  NumBalls.nSrazu:= frmOptions.RadioGroup1.ItemIndex+ 1;
  //число шариков, появляющихся после очередного хода:
  if NumBalls.nPotom<> frmOptions.RadioGroup2.ItemIndex+ 3 then
begin
    NumBalls.nPotom:= frmOptions.RadioGroup2.ItemIndex+ 3;
    //обновить шарики в табло:
    Pole.InitNextBalls;
    Pole.ShowNextBalls;
    dgNextBalls.Invalidate;
  end;

  //число шариков разного цвета:
  NumBalls.nColor:= frmOptions.RadioGroup3.ItemIndex+ 1;

  //так как параметры изменились, то обнулить рекорд:
  score.Rec:= 0;
end;
```


Табулятор имеет 3 страницы с компонентом *TRadioGroup* на каждой. Их нужно заполнить данными так, как показано выше.

Таким образом, по умолчанию сразу после начала игры появляются 5 шариков, но потом мы можем задать от 1 до 12. Чем больше шариков, тем энергичнее начинается игра!

После каждого хода на поле возникают ещё 3 шарика – по умолчанию, как в классичеком варианте. Если их количество уменьшить, то игра будет протекать очень вяло, но зато можно добавлять и *больше* шариков – до 9 за один раз. Ясно, что большое количество шариков сильно усложняет игру, и наоборот.

По умолчанию на поле появляются шарики 7 разных цветов. Ещё более увеличивать «пестроту» на поле смысла нет, поэтому число цветов можно только уменьшить – до одного. Так у игрока появятся новые возможности по составлению фигур из шариков!

Объединим всю информацию о шариках в *записи TNumBalls*:

```
type
  //количество шариков:
  TNumBalls= Record
    nSrazu: integer;
    nPotom: integer;
    nColor: integer;
  end;
```

```
var
  //число шариков в игре:
  NumBalls: TNumBalls;
```


Например, переменная *NumBalls.nSrazu* хранит число шариков, которые *сразу* появляются на поле.

При запуске приложения в процедуре *TForm1.FormCreate* как раз и устанавливаются значения по умолчанию:

```
  //шарики:
  NumBalls.nSrazu:= 5;
  NumBalls.nPotom:= 3;
  NumBalls.nColor:= 7;
```

Текст модуля *OptionsUnit:*

```
unit OptionsUnit;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
Dialogs,
  StdCtrls, ExtCtrls, ComCtrls;

type
  TfrmOptions = class(TForm)
    TabControl1: TTabControl;
    RadioGroup1: TRadioGroup;
    RadioGroup2: TRadioGroup;
    RadioGroup3: TRadioGroup;
    Button1: TButton;
    procedure TabControl1Change(Sender: TObject);
    procedure RadioGroup3Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
```

```
  end;

var
  frmOptions: TfrmOptions;

implementation

{$R *.DFM}

procedure TfrmOptions.TabControl1Change(Sender: TObject);
begin
  case TabControl1.TabIndex of
    0: begin
         RadioGroup1.Visible:= true;
         RadioGroup2.Visible:= False;
         RadioGroup3.Visible:= False;
       end;
    1: begin
         RadioGroup1.Visible:= False;
         RadioGroup2.Visible:= true;
         RadioGroup3.Visible:= False;
       end;
    2: begin
         RadioGroup1.Visible:= False;
         RadioGroup2.Visible:= False;
         RadioGroup3.Visible:= true;
       end;
  end;
end;

procedure TfrmOptions.RadioGroup3Click(Sender: TObject);
begin
  close
end;

procedure TfrmOptions.Button1Click(Sender: TObject);
begin
  close
end;

end.
```


Тут единственная «хитрость» состоит в том, чтобы показывать нужный набор переключателей, а остальные скрывать, чтобы они не мешали. Конечно, можно просто увеличить размеры формы *frmOptions* и разместить все переключатели (или движки, скроллеры и т.п.) так, чтобы они не перекрывали друг друга.

Ну и последнее, что можно изменять в игре, – это *размеры* игрового поля. Если вы помните, в классическом варианте размер поля постоянен – 9 х 9 клеток. Мы раздвинем границы – пусть в нашей программе размеры поля

изменяются от 6 x 6 до 12 x 12 клеток. После нажатия на кнопку *btnWH* под ней появится меню, в котором можно выбрать нужные размеры игрового поля (Рис. 5.22).

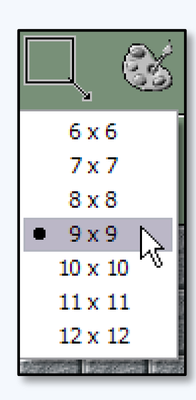


Рис. 5.22. Универсальное поле

Код процедуры аналогичен коду процедуры *TForm1.btnTypeFigClick*:

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЯ
procedure TForm1.btnWHClick(Sender: TObject);
var
  p: Tpoint;
begin
  //вывести всплывающее меню:
  p:= ClientToScreen(point(btnWH.Left, btnWH.Top+ btnWH.Height));
  popupmenu1.Popup(p.x, p.y);
  //сбросить рекорд:
  score.Rec:= 0;
end;
```

```
//ИЗМЕНИТЬ РАЗМЕРЫ ПОЛЯ В МЕНЮ
procedure TForm1.wh6Click(Sender: TObject);
var
  n: integer;
begin
  //показать пункт меню как выбранный:
  (Sender as TMenuItem).Checked:= True;
  //размеры поля:
  n:= (Sender as TMenuItem).Tag;
  //размеры поля:
  Pole.Width:= n;
  Pole.Height:= n;
  //изменить сетку:
  Prepare;
  dgPole.Invalidate;
  application.ProcessMessages;

  //начать новую игру:
  NewGame;
```

```
end;
```

Поэтому устанавливаем на форме компонент *TPopupMenu* и набираем строки (Рис. 5.23).

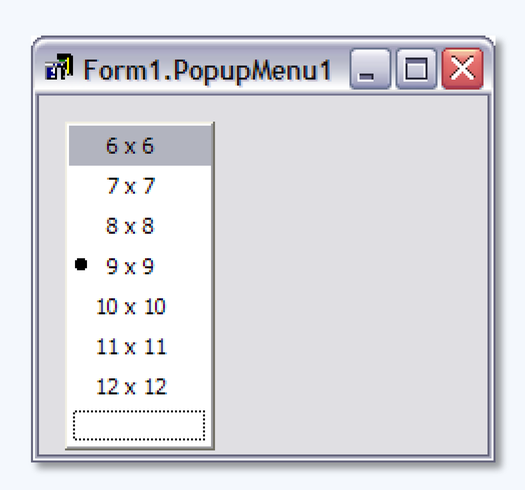


Рис. 5.23. Выбираем любимый размер

Затем задаём значения свойства **tag**=*6, 7,...,12.*

И последняя кнопка - *btnNewGame* - просто начинает новую игру по первому нашему требованию:

```
//НАЧАТЬ НОВУЮ ИГРУ
procedure TForm1.btnNewGameClick(Sender: TObject);
begin
  NewGame
end;
```

```
//НОВАЯ ИГРА
procedure TForm1.NewGame;
begin
  //подготовить сетку:
  Prepare;
  //начать игру:
  Pole.StartGame;
end;
```

Теперь мы умеем делать всё, кроме главного, - у нас нет игрового поля! Поэтому на компоненте *Panel1* мы установим сетку *dgPole*, размеры клеток которой - 32 x 32 пикселя:

```
 Var
 //размеры клеток сетки:
  wCell: integer = 32;
  hCell: integer = 32;
```

Этого вполне достаточно, чтобы картинки в них хорошо смотрелись на экране.

При старте каждой новой игры вызывается *процедура TForm1.Prepare*, в которой вычисляются размеры поля (то есть компонента *dgPole*) и оно устанавливается в центре панели:

```
//ПОДГОТОВИТЬ ИГРОВОЕ ПОЛЕ
procedure TForm1.Prepare;
var
  w,h, lw: integer;
begin
  //размер клетки в пикселях:
  w:= wCell;
  h:= wCell;
  //толщина линий:
  lw:= dgPole.GridLineWidth;
  //размеры игрового поля в клетках:
  dgPole.ColCount:= Pole.Width;
  dgPole.RowCount:= Pole.Height;
  //размеры в пикселях видимой части игрового поля:
  dgPole.Width:= 3 + (w + lw)* dgPole.ColCount-1;
  dgPole.Height:= 3 + (h + lw)* dgPole.RowCount-1;
  //отцентрировать поле:
  dgPole.Left:= panel1.Left+ (panel1.Width-dgPole.Width) div 2+1;
  dgPole.Top:= panel1.Top+ (panel1.Height -dgPole.Height) div 2+1;
end;
```


Так выглядит интерфейс программы в окне *Конструктора формы* (Рис. 5.24).

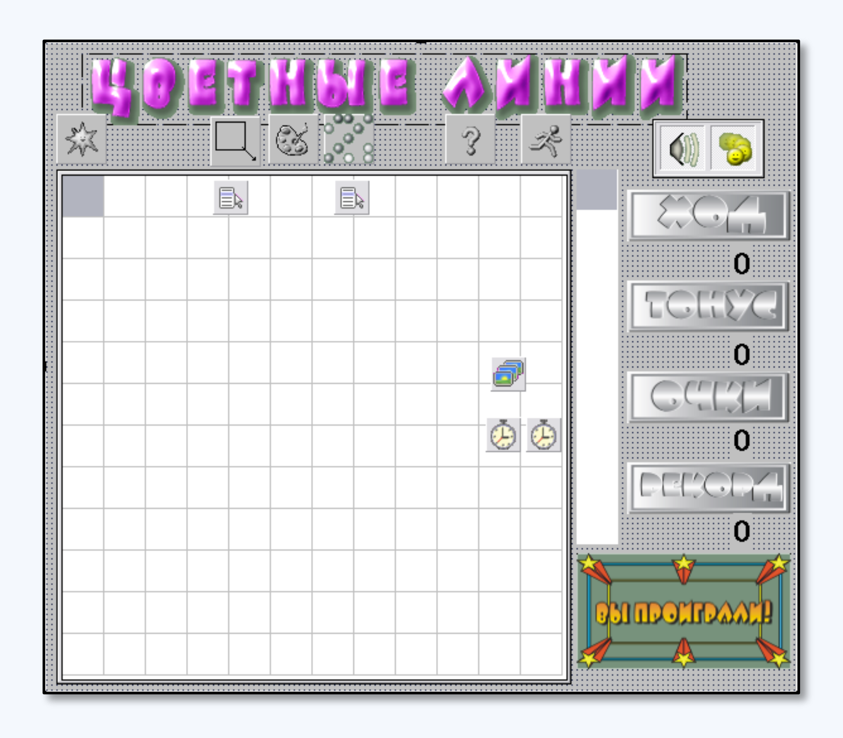


Рис. 5.24. Все в сборе!

А так – после запуска приложения (Рис. 5.25).

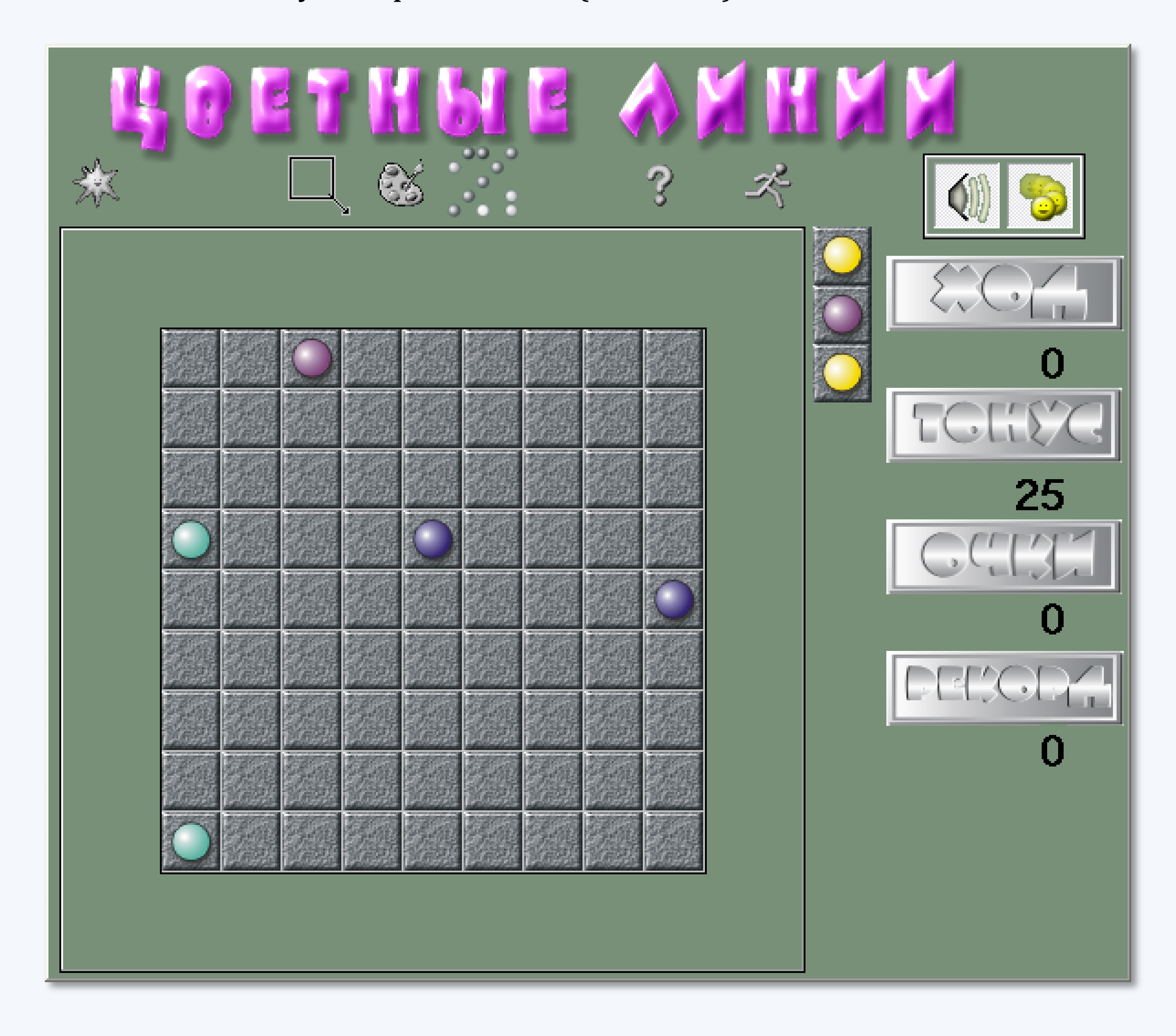


Рис. 5.25. Попрошу к шарам!

Как видите, есть несколько компонентов, ещё не охваченных нашим вниманием.

Но сейчас у нас есть дела поважнее – мы напишем класс *поля TPole*, для которого добавим к проекту новый *модуль* – *PoleUnit*. Это разумно, так как код достаточно длинный и лучше отделить его от основного текста программы. Тем более что этот модуль с небольшими изменениями можно будет использовать и в других проектах.

Введём *константы*:

```
const
  //макс. размеры поля:
  MAX_WIDTH=12;
  MAX_HEIGHT=12;
  //константы лабиринта:
```

```
  GO= -1;       // - проходимая клетка
  STOP= -2;     // - непроходимая клетка
  BEG_CELL= 0; // - начальная клетка пути
```

И новые *типы* данных:

```
type
  //цвет шариков:
  TBallColor= (bcNone, bcRed, bcGreen, bcWater, bcBrown, bcBlue,
bcLila, bcYellow);
  //цвет клетки поля:
  TCellColor= (ccGray, ccGreen, ccRed);
  //клетка поля:
  TCell= Record
    //цвет шарика в клетке:
    BallColor: TBallColor;
    //цвет клетки:
    CellColor: TCellColor;
    //новый шарик:
    New: boolean;
  end;
  //массив поля:
  TField= array[0..MAX_WIDTH, 0..MAX_HEIGHT] of integer;
```

Здесь всё настолько прозрачно, что любые комментарии излишни!

Сам класс поля мы разберём подробно:

```
  //Поле:
  TPole= class
    //ширина поля в клетках:
    Width: integer;
    //высота поля в клетках:
    Height: integer;
    //координаты выделенного шарика:
    SelBall: TPoint;
    //массив клеток:
    Cell: array[0..MAX_WIDTH, 0..MAX_HEIGHT] of TCell;
    //массив для поиска пути:
    masPath: TField;
    Path: array[0..MAX_WIDTH*MAX_HEIGHT] of TPoint;
    //следующие шарики:
    NextBalls: array[1..12] of TBallColor;

    //создать поле:
    constructor Create;
    //инициализация поля:
    procedure Init;
    //очистить поле:
    procedure Clear;
    //щелчок на поле:
```

```
  procedure Click(X, Y: integer);
  //инициализация первых шариков:
  procedure InitFirstBalls;
  //снять выделение с шарика:
  procedure DeActivateBall;
  procedure Animation;
  //начать новую игру:
  procedure StartGame;
  //появление шариков на поле:
  procedure CreateBalls;
  //загадать следующие шарики:
  procedure InitNextBalls;
  //показать следующие шарики в табло:
  procedure ShowNextBalls;

 protected

 private
  //минимальная длина цепочки:
  LenLines: integer;
  //минимальный размер блока:
  LenBlocks: integer;
  //задержка при уничтожении цепочки:
  DelayDestroy: integer;
  //задержка при возникновении шарика:
  DelayCreate: integer;
  //задержка при движении шарика:
  DelayMove: integer;

  //переместить шарик:
  procedure DoHod(X, Y: Integer);
  procedure DoMoveBall(X, Y: Integer);
  //выделить шарик:
  procedure ActivateBall(X, Y: Integer);
  //найти путь:
  function FindPath(EndCell: TPoint): boolean;
  //проложить путь:
  procedure CreatePath(EndCell: TPoint);
  //уничтожить фигуру:
  function DeleteFigs: integer;
  //подпрыгивающий шарик:
  procedure JumpBall;
  //игра закончена:
  procedure GameOver;
  procedure GameWin;
end;
```


Прежде всего, нужно создать поле; это делается при старте программы в процедуре *TForm1.FormCreate*:

```
  //создать поле:
  Pole:= tPole.Create;
```

При создании экземпляра класса мы задаём параметры игры по *умолчанию*:

```
///////////////////////////////////////////////////////////
///////////////////////// ПОЛЕ /////////////////////////////
///////////////////////////////////////////////////////////

constructor tPole.Create;
begin
  inherited create;
  //установить значения по умолчанию:
  LenLines:= 5;
  LenBlocks:= 6;
  DelayDestroy:= 150;
  DelayCreate:= 80;
  DelayMove:= 40;
end;
```


А именно: необходимая длина линий – 5 шариков, минимальное число шариков в блоке – 6; остальные переменные используются при анимации. После создания поля в процедуре *TForm1.FormActivate* запускается новая игра:

```
//ПОДГОТОВИТЬСЯ К ИГРЕ
procedure TForm1.FormActivate(Sender: TObject);
begin
  Prepare;
  delay(1000);
  Pole.StartGame;
end;
```

То же самое происходит и при нажатии на кнопку *Новая игра*:

```
//НОВАЯ ИГРА
procedure TForm1.NewGame;
begin
  //подготовить сетку:
  Prepare;
  //начать игру:
  Pole.StartGame;
end;
```

Старт новой игры – событие нетривиальное: здесь нужно всё *всесторонне обдумать*:

```
//ВЫВЕСТИ ПЕРВЫЕ ШАРИКИ
procedure TPole.StartGame;
begin
  //очистить поле:
  Clear;
```

```
  //показать следующие шарики в табло:
  InitNextBalls;
  ShowNextBalls;

  //разместить первые шарики в массиве:
  InitFirstBalls;
  //показать шарики на поле:
  CreateBalls;

  //оставшиеся ходы:
  tonus:= tTonus.Create;
  tonus.GetValue;
  tonus.ShowValue;

  //число ходов:
  Hod:= tHod.Create;
  Hod.ShowValue;

  //очки:
  Score.init;
  Score.ShowValue;

  //инициализация поля:
  Init;
  //снять выделение:
  DeActivateBall;

  sound('newgame');
  sleep(1500);
  application.ProcessMessages;
end;
```


Поскольку игровое поле представлено у нас и сеткой *dgPole* на форме, и массивом клеток *Cell: array[0..MAX_WIDTH, 0..MAX_HEIGHT] of TCell;*, то в массиве нужно убрать все шарики с поля, присвоив переменным *Pole.Cell[].BallColor* значение *bcNone*, что и означает отсутствие шариков в соответствующих клетках, а во все клетки сетки мы должны вывести картинку, изображающую *пустую* клетку:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TPole.Clear;
var
  i, j: integer;
  dr: TRECT;
begin
  with form1, dgPole do
    for j:= 0 to RowCount do
      for i:= 0 to ColCount do begin
        Pole.Cell[i, j].BallColor:= bcNone;
        dr:= CellRect(i, j);
        imlPic.Draw(Canvas, dr.left, dr.top, 0);
```

```
        end;
end;
```

Ну, если с массивом всё ясно, то манипуляции с полем ещё требуют пояснений.

Картинки с клетками поля во всех состояниях хранятся в компоненте *imlPic: TImageList* (Рис. 5.26). Их огромное количество, что объясняется анимацией.

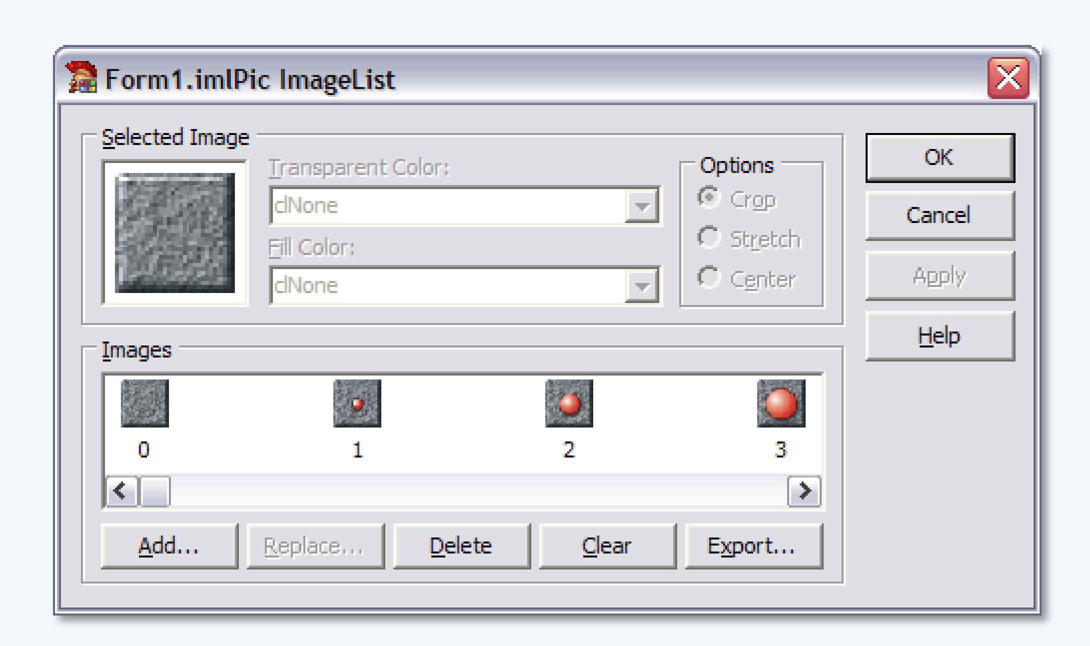


Рис. 5.26. Шарики в списке картинок

Все они имеют размеры 32 х 32 пикселя, чтобы точно соответствовать клеткам поля. Конечно, требуется некоторая графическая сноровка, чтобы их нарисовать (Рис. 5.27).

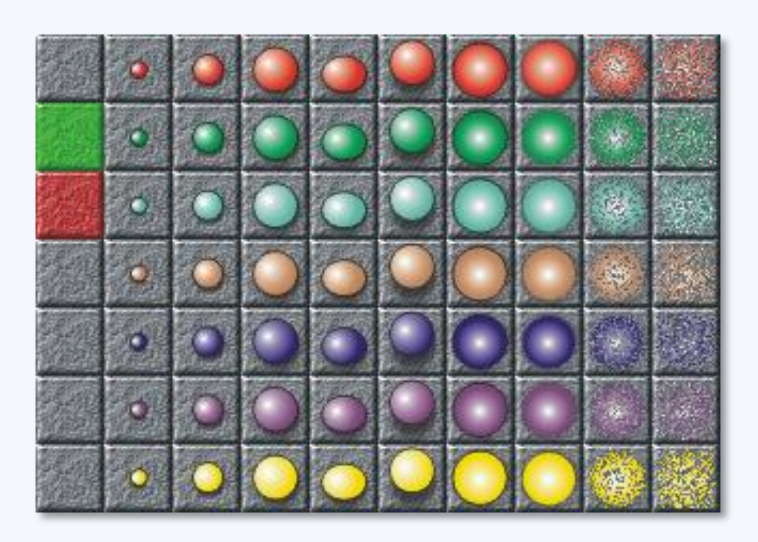


Рис. 5.27. Кадры анимации шариков

Лично я пользовался векторным редактором *CorelDRAW*, чего, как говорится, и вам желаю.

Когда картинки уже загружены в *imlPic*, то вы вести их в нужную клетку поля не представляет ни малейшего труда:

```
dr:= CellRect(i, j);
imlPic.Draw(Canvas, dr.left, dr.top, 0);
```

Так как в компоненте *TDrawGrid* картинки «не держатся», то их необходимо постоянно перерисовывать при изменении ситуации на поле:


```
//ОТРИСОВАТЬ СЕТКУ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow:
Integer;
  Rect: TRect; State: TGridDrawState);
var
  r, dr: TRECT;
  index: integer;
begin
  //размеры картинок:
  dr:= Bounds(Rect.Left, Rect.Top, wCell, hCell);
  //картинка с шариком в ячейке:
  index:= (ord(Pole.Cell[ACol, ARow].BallColor));
  if index> 0 then //- в клетке есть шарик
    index:= (index-1)* period + 3
  else             //- пустая клетка
    index:= (ord(Pole.Cell[ACol, ARow].CellColor))* period;
  //скопировать картинку из списка в ячейку:
  imlPic.Draw(dgPole.Canvas, dr.left, dr.top, index);
end;
```

Константа *PERIOD* равна числу картинок при анимации шарика:

```
const
  //число картинок на шарик одного цвета:
  PERIOD= 10;
```

При инициализации поля в *процедуре tPole.Init* все клетки поля (не шарики!) окрашиваются в *серый* цвет, который они сохраняют, пока ни один шарик на поле не выделен:

```
//ИНИЦИАЛИЗАЦИЯ
procedure tPole.Init;
var i, j: integer;
begin
  //все клетки - серые:
  for j:= 0 to Height do
    for i:= 0 to Width do
```

```
      Cell[i,j].CellColor:= ccGray;

  //нет выделенного шарика:
  SelBall:= Point(-1,-1);
end;
```

В переменной *SelBall* хранятся координаты того шарика, на котором игрок кликнул. Так как координаты поля начинаются с *нуля*, то *отрицательные* значения говорят о том, что выделенного шарика на поле нет.

Затем в процедуре *StartGame* заполняется массив *NextBalls*, в котором хранятся шарики, которые появятся на поле после первого хода игрока:

```
//ИНИЦИАЛИЗАЦИЯ СЛЕДУЮЩИХ ШАРИКОВ
procedure TPole.InitNextBalls;
var
  i: integer;
begin
  Randomize;
  for i:= 1 to NumBalls.nPotom do begin
    NextBalls[i]:= TBallColor(ord(random(NumBalls.nColor)+1));
  end
end;
```

Также их нужно показать в сетке *dgNextBalls*, которая расположена справа от игрового поля:

```
//ПОКАЗАТЬ СЛЕДУЮЩИЕ ШАРИКИ В ТАБЛО
procedure TPole.ShowNextBalls;
var
  i: integer;
  dr: TRECT;
  index: integer;
begin
  for i:= 1 to NumBalls.nPotom do begin
    index:= (ord(NextBalls[i])-1) * period;
    dr:= form1.dgNextBalls.CellRect(0, i-1);
    form1.imlPic.Draw(form1.dgNextBalls.Canvas, dr.left, dr.top,
index+3);
  end;
end;
```

Процедура отрисовки клеток этой сетки:

```
//////////////////////////////////////////////////////////////
////////////////////// СЕТКА-ТАБЛО //////////////////////////
//////////////////////////////////////////////////////////////

procedure TForm1.dgNextBallsDrawCell(Sender: TObject; ACol, ARow:
Integer;
```

```
  Rect: TRect; State: TGridDrawState);
var
  dr: TRECT;
  index: integer;
begin
  dr:= dgNextBalls.CellRect(0,ARow);
  //стереть ячейки без шариков:
  if ARow+1 > NumBalls.nPotom then begin
    dgNextBalls.Canvas.Brush.Color:= fon;
    dgNextBalls.Canvas.FillRect(dgNextBalls.CellRect(ACol, ARow))
  end
  else begin
  //вывести шарик в ячейке:
  index:= (ord(Pole.NextBalls[ARow+1])-1) * period;
  imlPic.Draw(dgNextBalls.Canvas, dr.left, dr.top, index+3);
  end;
end;
```


Каждый программист должен предусмотреть в программе *уловки*, помогающие в игре. Мы сделаем так:

```
//ХИТРЫЙ ПРИЁМ
procedure TForm1.dgNextBallsMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ACol, ARow: integer;
begin
  //координаты клетки:
  dgNextBalls.MouseToCell(X, Y, ACol, ARow);
  //если нажать кнопку мыши на первом шарике,
  //то все след. шарики заменятся на новые:
  if ARow= 0 then begin
    //показать следующие шарики в табло:
    Pole.InitNextBalls;
    Pole.ShowNextBalls;
  end
  //если нажать кнопку мыши на втором шарике,
  //то поле очистится:
  else if ARow= 1 then begin
    Pole.Clear;
    //снять выделение:
    Pole.DeActivateBall;
    //показать следующие шарики в табло:
    Pole.InitNextBalls;
    Pole.ShowNextBalls;

    //разместить первые шарики в массиве:
    Pole.InitFirstBalls;
    //появление шариков на поле с бульканием:
    Pole.CreateBalls;

    //осталось ходов:
    tonus:= tTonus.Create;
```

```
    tonus.GetValue;
    tonus.ShowValue;
  end;
end;
```

То есть: нажимая *первый* (верхний) шарик в сетке-табло, мы получаем *новый* набор следующих шариков и можем выбрать нужную нам комбинацию. Следующий приём ещё более радикальный. Нажав *второй* шарик, мы полностью очистим поле, сохранив при этом все свои честно заработанные очки!

Немного нелогично, но после *следующих* шариков, мы заполняем массив *Cell текущих* шариков (тех, что *сразу* появляются на поле):

```
//ИНИЦИАЛИЗАЦИЯ ПЕРВЫХ ШАРИКОВ
procedure TPole.InitFirstBalls;
var
  i, j, n: integer;
begin
  //разместить первые шарики в массиве:
  n:= 0;
  repeat
    i:= random(Width);
    j:= random(Height);
    if Cell[i, j].BallColor= bcNone then begin
      //поместить в пустую клетку шарик случайного цвета:
      Cell[i, j].BallColor:=
TBallColor(ord(random(NumBalls.nColor)+1));
      Cell[i, j].new:= true;
      inc(n)
    end;
  until n= NumBalls.nSrazu;
end;
```

Для этого мы выбираем *случайную* клетку поля и, если она *пустая*, то помещаем в неё шарик опять же *случайного* цвета (из возможных), а также указываем, что шарик «новый»:

```
Cell[i, j].new:= true;
```

В следующей процедуре:

```
//ПОЯВЛЕНИЕ ШАРИКОВ:
procedure TPole.CreateBalls;
var
  i, j, index: integer;
  dr: TRECT;
begin
  for j:= 0 to Height-1 do
```

```
    for i:= 0 to Width-1 do begin
      dr:= form1.dgPole.CellRect(i, j);
      index:= (ord(Pole.Cell[i, j].BallColor)-1) * period;
      if Cell[i, j].new= true then begin
        Cell[i, j].new:= false;
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+1);
        Delay(DelayCreate);
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+2);
        Delay(DelayCreate);
        Sound('buljk');
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+ 3);
      end;
    end;
end;
```


мы просматриваем всё поле и, найдя «новый» шарик, выводим его на поле с анимацией и звуковым эффектом.

Здесь нам протребуются две простенькие процедуры. Первая издаёт звуки (файлы хранятся в *папке WAV*):

```
//ВОСПРОИЗВЕСТИ ЗВУК
procedure Sound(name: PChar);
var s: PChar;
begin
  if form1.sbtSound.down= true then begin
    //сформировать имя файла:
    s:= PChar('wav\'+ name + '.wav');
    sndPlaySound(s, SND_ASYNC or SND_FILENAME);
  end;
end;
```

Тут следует заметить, что булькающие звуки, сопровождающие появление шариков на свет божий, могут и раздражать игрока, поэтому добавим на форму *кнопку sbtSound* (**Hint**=Звук вкл-откл **AllowAllUp**=True **Down**=True **GroupIndex**=1). Она действует как *переключатель*. Когда кнопка находится в *нажатом* положении (по умолчанию это так и есть, то есть мы надеемся, что игроку приятен наш выбор звуков), звук воспроизводится, когда в *отжатом* – нет. Всё демократично…

Так как шарики возникают в сетке излишне резво, то все фазы их рождения разделены паузой:

```
//ЗАДЕРЖКА
procedure Delay(MSecs: Longint);
```

```
var
  FirstTickCount, Now: Longint;
begin
  FirstTickCount := GetTickCount;
  repeat
    Application.ProcessMessages;
    Now := GetTickCount;
  until (Now - FirstTickCount >= MSecs) or (Now < FirstTickCount);
end;
```

При создании поля устанавливается такая *задержка*:


```
DelayCreate:= 80;
```

Вы можете изменить её в полном соответствии со своим темпераментом…

Идём дальше. В каждой игре важна *статистика*, без которой играть неинтересно! В нашей игре - это количество оставшихся у нас ходов:

```
  //оставшиеся ходы:
  tonus:= tTonus.Create;
  tonus.GetValue;
  tonus.ShowValue;
```

Число уже сделанных ходов:

```
  //число ходов:
  Hod:= tHod.Create;
  Hod.ShowValue;
```

Набранные очки:

```
  //очки:
  Score.init;
  Score.ShowValue;
```

Заведём для статистики ещё три *класса*: *tTonus*, *tHod* и *tScore* в главном модуле программы. Затем объявим соответствующие *переменные*:

```
var
  //игровой тонус:
  Tonus: tTonus;
  //число ходов:
  Hod: tHod;
  //набранные очки:
  Score: tScore;
```

Рассмотрим все классы по очереди. *Игровой «тонус»* :

```
  //игровой тонус:
  type tTonus= class
    //число оставшихся ходов:
    value: integer;
    //число пустых клеток на поле:
    EmptyCell: integer;
    //есть ли на поле шарики?
    EmptyPole: boolean;
    //есть ли на поле хотя бы одна свободная клетка?
    FullPole: boolean;
    constructor Create;
    //показать:
    procedure ShowValue;
    //получить значение тонуса:
    procedure GetValue;

  end;
```


Здесь нас в, первую очередь, может интересовать, сколько ходов мы сможем ещё продержаться – *value*. Определить их число можно так: подсчитаем оставшиеся пустые клетки *EmptyCell* и разделим их на число шариков, появляющихся после каждого хода - *NumBalls.nPotom*. Значение «тонуса» выводим для всеобщего обозрения в метку *lblTonus* (Рис. 5.28).



Рис. 5.28. Тонус отсутствует!

В компонент *TImage* мы загружаем картинку с подходящей надписью (все картинки находятся в *папке image*).

```
///////////////////////////////////////////////////////////
//////////////////// ИГРОВОЙ ТОНУС ////////////////////////
///////////////////////////////////////////////////////////

constructor tTonus.Create;
begin
  inherited;
  value:= 0;
end;

//ВЫЧИСЛИТЬ КОЛИЧЕСТВО ОСТАВШИХСЯ ХОДОВ
procedure tTonus.GetValue;
var
  i, j: integer;
begin
```

```
  //подсчитать число пустых клеток на поле:
  EmptyCell:= 0;
  for j:= 0 to Pole.Height-1 do
    for i:= 0 to Pole.Width-1 do
      if Pole.Cell[i, j].BallColor= bcNone then inc(EmptyCell);
  //число оставшихся ходов:
  value:= EmptyCell div NumBalls.nPotom;
  //проверить, не пусто ли поле:
  if EmptyCell = Pole.Height * Pole.Width then
    EmptyPole:= true
  else
    EmptyPole:= false;
  //проверить, есть ли на поле свободные клетки:
  if EmptyCell = 0 then
    FullPole:= true
  else
    FullPole:= false;
end;

//ВЫВЕСТИ ЗНАЧЕНИЕ ТОНУСА НА ЭКРАН
procedure tTonus.ShowValue;
begin
  form1.lblTonus.Caption:= inttostr(value);
  form1.lblTonus.Invalidate;
  application.ProcessMessages
end;
```


Во всех играх важно знать *число* сделанных *ходов*:

```
  //ходы:
  type tHod= class
    //число сделанных ходов:
    value: integer;
    constructor Create;
    //показать:
    procedure ShowValue;
  end;
```

Здесь всё просто:

```
///////////////////////////////////////////////////////////
//////////////////////// ХОДЫ /////////////////////////////
///////////////////////////////////////////////////////////

constructor tHod.Create;
begin
  inherited;
  value:= 0;
end;

//ВЫВЕСТИ ЗНАЧЕНИЕ НА ЭКРАН
procedure tHod.ShowValue;
```

```
begin
  form1.lblHod.Caption:= inttostr(value);
  form1.lblHod.Invalidate;
  application.ProcessMessages
end;
```

Число ходов мы показываем в метке *lblHod* (Рис. 5.29).



Рис. 5.29. Пора ходить!

Ещё более важно знать *набранные очки*, чтобы установить *рекорд*:

```
  //очки:
  type tScore= class
    //набранные очки:
    value: integer;
    //рекорд:
    Rec: integer;
    constructor Create;
    procedure Init;
    //показать:
    procedure ShowValue;
    //установить новое значение:
    procedure SetValue(n: integer);
    //запомнить новый рекорд:
    procedure SetRec;
  end;
```

```
////////////////////////////////////////////////////////////
//////////////////////// ОЧКИ //////////////////////////////
////////////////////////////////////////////////////////////

constructor tScore.Create;
begin
  inherited;
end;

//ИНИЦИАЛИЗАЦИЯ
procedure tScore.Init;
begin
  value:= 0;
end;

//ПОДСЧИТАТЬ НАБРАННЫЕ ОЧКИ
procedure tScore.SetValue(n: integer);
```

```
begin
  if n> 0 then
    if TypeFig= tfLines then
      value:= value + 5*n - 20
    else
      value:= value + 6*n - 26
end;

//ВЫВЕСТИ ОЧКИ НА ЭКРАН
procedure tScore.ShowValue;
begin
  with form1 do begin
    lblScore.Caption:= inttostr(value);
    lblScore.Invalidate;
    lblRecord.Caption:= inttostr(rec);
    lblRecord.Invalidate;
  end;
  application.ProcessMessages
end;

//РЕКОРДНЫЕ ОЧКИ
procedure tScore.SetRec;
begin
  if value> rec then
    rec:= value;
  showValue;
end;
```


Набранные очки и рекорд мы печатаем в метках *lblScore* и *lblRecord,* соответственно (Рис. 5.30).





Рис. 5.30. Текущие очки и рекорд

В игре, конечно, очень важен подсчёт набранных очков. Я остановился на таких формулах:

```
//ПОДСЧИТАТЬ НАБРАННЫЕ ОЧКИ
procedure tScore.SetValue(n: integer);
begin
  if n> 0 then
    if TypeFig= tfLines then
      value:= value + 5*n - 20
    else
      value:= value + 6*n - 26
end;
```

Вы можете выбрать формулы по своему вкусу!

И - нам осталось рассмотреть самый конец процедуры *TPole.StartGame*:

```
  //инициализация поля:
  Init;
  //снять выделение:
  DeActivateBall;

  sound('newgame');
  sleep(1500);
  application.ProcessMessages;
```


Процедура *Init* нам уже известна, а *процедура DeActivateBall* просто снимает выделение с шарика:

```
//СНЯТЬ ВЫДЕЛЕНИЕ С ШАРИКА
procedure tPole.DeActivateBall;
var
  i, j: integer;
begin
  SelBall:= Point(-1,-1);
  //остановить таймер:
  form1.timer2.Interval:= 1;
  form1.timer2.Enabled:= false;

  //все клетки - серые:
  for j:= 0 to Height-1 do
    for i:= 0 to Width-1 do
      Cell[i,j].CellColor:= ccGray;
  form1.dgPole.Invalidate;
  application.ProcessMessages;
end;
```

Сейчас мы проясним ситуацию с шариками на поле до полной прозрачности.

Мы находимся в положении, когда всё готово для новой игры: имеется поле заданных размеров, на нём находится некоторое количество шариков, нам известны следующие шарики, мы имеем всю информацию о ходе игры. Что должен сделать игрок дальше? – Конечно, выбрать какой-нибудь шарик на поле и указать место, на которое он должен переместиться!

Итак, нажимаем кнопку мышки на клетке поля:

```
//НАЖАТЬ КНОПКУ МЫШИ НА ПОЛЕ
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button: TMouseButton;
```

```
  Shift: TShiftState; X, Y: Integer);
var
  ACol, ARow: integer;
begin
  //координаты клетки:
  dgPole.MouseToCell(X, Y, ACol, ARow);
  Pole.Click(ACol, ARow);
end;
```

Обработка этого события происходит в *процедуре tPole.Click:*


```
//ЩЕЛЧОК НА ПОЛЕ
procedure tPole.Click(X, Y: integer);
//X, Y - координаты клетки
begin
  sound('click');
  //если кнопка нажата в клетке с шариком, выделить его:
  if Cell[X, Y].BallColor <> bcNone then
    ActivateBall(X, Y)
  //если кнопка нажата в пустой клетке, то сделать ход:
  else
    DoHod(X, Y)
end;
```

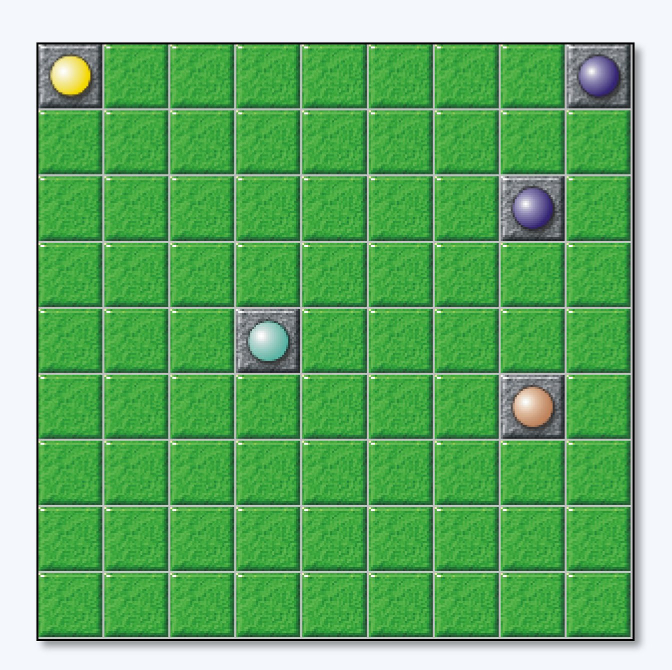
Здесь нужно проверить, есть ли шарик в этой клетке. Если есть, мы «активируем» (выделяем) его. Если же клетка *пустая*, то нужно выполнить ход.

Рассмотрим первую процедуру - *ActivateBall(X, Y):*

```
//ВЫДЕЛИТЬ ШАРИК
procedure tPole.ActivateBall(X, Y: Integer);
//X, Y - координаты клетки с шариком
var
  i,j: integer;
begin
  //если шарик повторно выделен:
  if  (X= SelBall.x) and (Y= SelBall.y) then
    DeActivateBall
  //если выделен новый шарик -
  else begin
    //запомнить координаты шарика:
    SelBall:= Point(X, Y);
    //запустить прыжки шарика:
    JumpBall;
    //выделить доступные клетки:
    FindPath(Point(-1,-1));
      for j:= 0 to height-1 do
        for i:= 0 to width-1 do
          //если в клетке число >0, то клетка доступна:
          if masPath[i,j]> 0 then
            Cell[i,j].CellColor:= ccGreen
          //иначе - клетка недоступна:
          else
```

```
            Cell[i,j].CellColor:= ccRed;
    form1.dgPole.Invalidate;
  end;
end;
```

Поскольку на поле нужно кликать в *двух* клетках, то мы должны как-то отмечать кликнутый шарик, чтобы не потерять его из виду, делая второй клик. Сделать это можно по-разному. Я выбрал простую анимацию выделенного шарика – он *прыгает* (Рис. 5.31) в своей клетке (к сожалению, на статичной картинке этого не покажешь).



Выделен шарик в третьей сверху строке

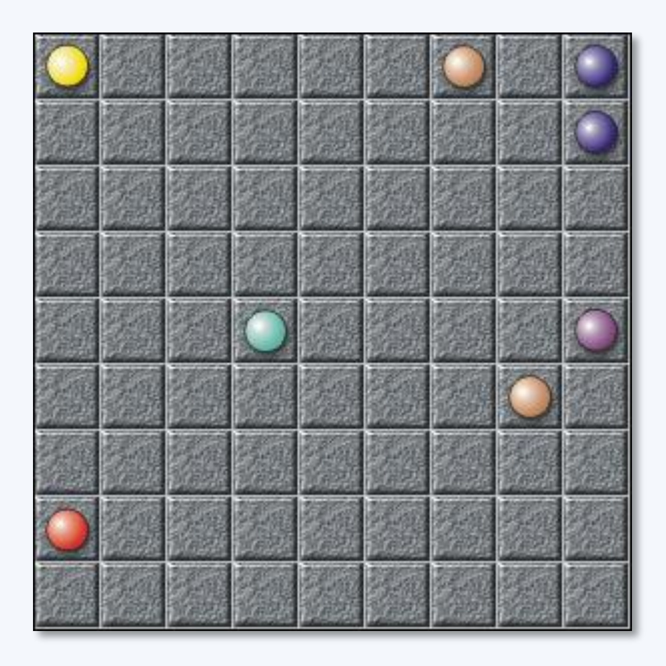


Рис. 5.31. Ход сделан – выделенный шарик переместился в новую клетку

В игре бывает всякое: вы могли по ошибке кликнуть не тот шарик. Тогда кликните на нём ещё раз – и он станет *обычным* шариком:

```
  //если шарик повторно выделен:
  if  (X= SelBall.x) and (Y= SelBall.y) then
    DeActivateBall
```

Если клик удачный, то мы запоминаем координаты клетки и заставляем шарик прыгать:


```
    //запомнить координаты шарика:
    SelBall:= Point(X, Y);

    //запустить прыжки шарика:
    JumpBall;
```

Так как анимация - процесс весьма сложный, то заведём новый тип данных *TAnimate*:

```
//анимация шариков:
  TAnimate= Record
   //интервал:
   Interval: cardinal;
   //координаты клетки поля:
   x, y: integer;
   //список индексов в списке:
   index: array[1..100] of integer;
   //текущий индекс в массиве:
   ptrIndex: integer;
   //число картинок в анимации:
   Frame: integer;
   //имя файла звука:
   sound: PChar;
   //индекс для воспроизведения звука:
   SoundIndex: integer;
   //число повторений:
   Rep: integer;
   //выполнено:
   Exec: integer;
  end;
```

И соответствующую *переменную*:

```
var
  //анимированная ячейка:
  AnimateCell: TAnimate;
```

После клика на шарике мы задаём параметры анимации:

```
//ВЫДЕЛИТЬ ШАРИК
```

```
procedure tPole.JumpBall;
var i: integer;
begin
    //запустить прыжки шарика:
    AnimateCell.Interval:= 120;
    AnimateCell.x:= SelBall.x;
    AnimateCell.y:= SelBall.y;
    i:= (ord(Cell[SelBall.x, SelBall.y].BallColor)-1) * period;
    AnimateCell.index[1]:=i+4;
    AnimateCell.index[2]:=i+3;
    AnimateCell.index[3]:=i+5;
    AnimateCell.index[4]:=i+3;
    AnimateCell.ptrIndex:= 1;
    AnimateCell.Frame:= 4;
    AnimateCell.sound:= 'jump1';
    AnimateCell.SoundIndex:= 4;
    AnimateCell.Rep:= 0;

end;
```


Анимация шарика реализована так. Через заданный интервал времени *Interval* в заданную клетку *(x, y)* последовательно выводятся картинки из компонента с картинками *imlPic*, который мы уже рассматривали. Наиболее естественно менять картинки с помощью компонента *таймер Timer2* (**Interval**=1 **Enabled**=False)

Как только он будет запущен (*form1.timer2.Enabled:= true;*), вызывается процедура

```
//ТАЙМЕР-АНИМАТОР:
procedure TForm1.Timer2Timer(Sender: TObject);
begin
  //запустить подпрыгивающий шарик:
  Pole.Animation;
end;
```

А она, в свою очередь, через заданный промежуток времени вызывает процедуру *TPole.Animation:*

```
//ПОДПРЫГИВАЮЩИЙ ШАРИК
procedure TPole.Animation;
//x. y - координаты клетки поля;
//index - номер картинки;
var
  dr: Trect;
  x,y, index: integer;
begin
  //все повторения выполнены:
```

```
  if (AnimateCell.Rep<> 0) and (AnimateCell.Exec= AnimateCell.Rep)
then
  begin
    form1.timer2.Enabled:= false;
    exit;
  end;
  form1.timer2.Interval:= AnimateCell.Interval;
  x:= AnimateCell.x;
  y:= AnimateCell.y;
  //квадрат клетки:
  dr:= form1.dgPole.CellRect(x,y);
  //номер текущей картинки:
  index:= AnimateCell.Index[AnimateCell.ptrIndex];
  //вывести текущую картинку:
  form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top, Index);

  if (AnimateCell.sound<> '') and
    (AnimateCell.SoundIndex= AnimateCell.ptrIndex)
  then
    Sound(AnimateCell.sound);

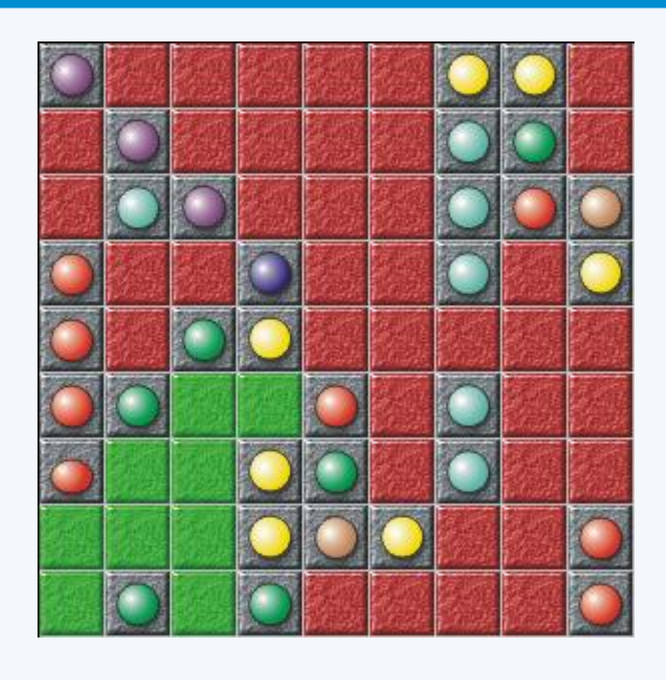
  inc(AnimateCell.ptrIndex);
  if AnimateCell.ptrIndex> AnimateCell.Frame then
    AnimateCell.ptrIndex:= 1;

  if AnimateCell.rep<> 0 then
    inc(AnimateCell.exec);
end;
```


Тут всё несколько заковыристо, но это свойство всех анимаций!

Этим можно было бы и ограничиться, но давайте поможем игроку, ведь ему будет значительно легче ориентироваться на поле, если мы *отметим* клетки, в которые он *может* сделать ход (Рис. 5.32).

```
    //выделить доступные клетки:
    FindPath(Point(-1,-1));
      for j:= 0 to height-1 do
        for i:= 0 to width-1 do
          //если в клетке число >0, то клетка доступна:
          if masPath[i,j]> 0 then
            Cell[i,j].CellColor:= ccGreen
          //иначе - клетка недоступна:
          else
            Cell[i,j].CellColor:= ccRed;
```



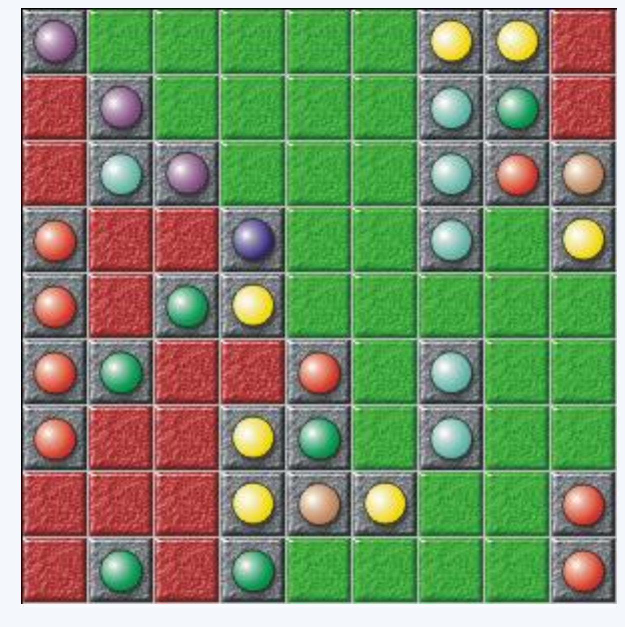


Рис. 5.32. Выделен *красный* шарик в 7-й строке Выделен *жёлтый* шарик в 4-й строке

Эти рисунки показывают, как изменяется цвет клеток на поле после клика на шарике. *Серые* в нормальном состоянии клетки окрашиваются в *зелёный* – они доступные – и в *красный* – недоступные – цвет. Согласитесь, теперь выбрать нужный ход значительно проще!

Как найти путь в лабиринте, мы уже разобрались в начале главы. Там же мы отмечали, что этот алгоритм позволяет и закрашивать замкнутые области. Для этого достаточно указать координаты конечной клетки так, чтобы она была заведомого за пределами поля:

```
FindPath(Point(-1,-1));
```

Путь, естественно, функция *TPole.FindPath* путь не найдёт, но зато она отметит все клетки, доступные из начальной (в данном случае это клетка с выделенным шариком). В массиве *masPath* им соответствуют *ненулевые* значения:

```
//НАЙТИ ПУТЬ МЕЖДУ ДВУМЯ КЛЕТКАМИ
function TPole.FindPath(EndCell: TPoint): boolean;
var
  //список координат:
  CoordList: array[1..MAX_WIDTH * MAX_HEIGHT] of TPoint;
  //указатели в списке:
  ptrWrite, ptrRead: integer;
  p, q: integer;
  i, j: integer;

  //проверить координаты
  function TestCoord(x,y: integer): boolean;
```

```
  begin
    Result:= true;
    if (x<0) or (x> Width-1) or (y<0) or (y> Height-1)or
     (masPath[x,y]<> GO) then Result:= false;
  end;

begin
  //формируем массив лабиринта:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      //если в клетке есть шарик, то клетка непроходима:
      if Cell[i,j].BallColor<> bcNone then
        masPath[i,j]:= STOP
      //иначе - клетка непроходима:
      else
        masPath[i,j]:= GO;

  //заносим в список координаты начальной клетки:
  CoordList[1]:= SelBall;
  //в начальную клетку в массиве CellPath ставим BEG_CELL= 0:
  masPath[SelBall.x,SelBall.y]:= BEG_CELL;
  //устанавливаем указатель для считывания координат на начало
списка:
  ptrRead:= 1;
  //устанавливаем указатель для записи новых координат на
следующий индекс:
  ptrWrite:= 2;

  //двигаемся от начала списка к его концу, пока он не кончится:
  while ptrRead < ptrWrite do begin
    //координаты текущей клетки:
    p:= CoordList[ptrRead].x; q:= CoordList[ptrRead].y;
    //проверяем соседние с ней клетки:
    for i:= p - 1 to p + 1 do
      for j:= q - 1 to q + 1 do
        //если нашли соседнюю проходимую клетку,
        if ((i=p) or (j=q)) and TestCoord(i,j) then
        begin
          //то записываем в неё число, на единицу большее,
          //чем в текущей клетке:
          masPath[i,j]:= masPath[p,q] + 1;

          //если дошли до конечной клетки,
          if (i= EndCell.x) and (j= EndCell.y) then begin
            //то путь найден:
            Result:= True;
            exit;
          end
          else begin
            //записываем координаты соседней клетки в конец
списка:
            CoordList[ptrWrite]:= Point(i,j);
            //перемещаем указатель:
```

```
            inc(ptrWrite);
          end;
        end;
      //переходим к следующей клетке в списке:
      inc(ptrRead);
  end;
  //путь не найден:
  Result:= False;
end;
```

После формирования массива *masPath* нам достаточно просмотреть все клетки поля и окрасить их в соответствии со значением *Cell[].CellColor*.

Итак, игрок выделил шарик, а затем указал клетку, в которую его следует переместить:

```
//ПЕРЕМЕСТИТЬ ШАРИК
procedure tPole.DoHod(X, Y: Integer);
//X, Y - координаты клетки
var
  n, i, j: integer;
begin
  //если шарик не был выделен, то перемещать нечего:
  if (SelBall.x= -1) and(SelBall.y= -1) then begin
    Sound('cantmove'); exit
  end;

  //найти путь в новую клетку -

  //если пути нет, то переместить шарик нельзя:
  if not FindPath(Point(x,y)) then begin // - пути нет
    Sound('cantmove'); exit
  end;

  //если путь найден - переместить шарик:
  DoMoveBall(x, y);

  //увеличить число сделанных ходов:
  inc(Hod.value);
  //и показать на экране:
  Hod.ShowValue;

  //если образовалась нужная фигура, уничтожить её:
  n:= DeleteFigs;

  //вычислить и показать набранные очки:
  Score.SetValue(n);
  Score.ShowValue;

  //вычислить и показать игровой тонус:
  Tonus.GetValue;
```

```
Tonus.ShowValue;

//если был сделан результативный ход и на поле есть шарики,
//то ждать следующего хода:
if (n> 0) and (Tonus.EmptyPole= false) then exit;

//вывести следующие шарики NextBalls -

//если для новых шариков не осталось места, то игра проиграна:
if Tonus.EmptyCell < NumBalls.nPotom then begin
  GameOver;
  exit
end;

//если поле пустое - победа:
if Tonus.EmptyPole= true then begin
  GameWin;
  exit
end;

//найти место для новых шариков:
n:= 0;
repeat
  i:= random(Width);
  j:= random(Height);
  if Cell[i, j].BallColor= bcNone then begin
    //поместить в пустую клетку «следующий» шарик:
    Cell[i, j].BallColor:= NextBalls[n+1];
    Cell[i, j].new:= true;
    inc(n)
  end;
until n= NumBalls.nPotom;
//появление шариков с бульканием:
CreateBalls;

//если образовалась нужная фигура, уничтожить её:
n:= DeleteFigs;

//вычислить и показать набранные очки:
Score.SetValue(n);
Score.ShowValue;

//вычислить и показать игровой тонус:
Tonus.GetValue;
Tonus.ShowValue;

//показать следующие шарики:
InitNextBalls;
ShowNextBalls;

//если поле пустое - победа:
if Tonus.EmptyPole= true then begin
  GameWin;
```

```
    exit
  end;

  //если ходов нет - поражение:
  if Tonus.FullPole= true then begin
    GameOver;
    exit
  end;
end;
```

Сначала мы должны учесть «несчастный случай», когда игрок кликнул на *пустой* клетке поля, не выделив шарик. Понятно, что ничего хорошего в этом случае делать не нужно:


```
  //если шарик не был выделен, то перемещать нечего:
  if (SelBall.x= -1) and (SelBall.y= -1) then begin
    Sound('cantmove'); exit
  end;
```

Если игрок действует правильно, то мы ищем путь из начальной клетки в конечную:

```
  //если пути нет, то переместить шарик нельзя:
  if not FindPath(Point(x,y)) then begin // - пути нет
    Sound('cantmove'); exit
  end;
```

Хоть мы и позаботились о раскрашивании клеток поля в разные цвета, игрок может оказаться невнимательным и указать *недоступную* клетку. Если же путь найден, мы перемещаем шарик:

```
  //если путь найден - переместить шарик:
  DoMoveBall(x, y);
```

Занятие это не из простых:

```
//ПЕРЕМЕСТИТЬ ШАРИК
procedure tPole.DoMoveBall(X, Y: Integer);
//X, Y - координаты клетки
var
  dr: TRECT;
  index, i,j: integer;
  bc: TBallColor;
  p, q: integer;
begin
  //убрать выделение с клеток:
  DeactivateBall;
  //создать путь:
  CreatePath(Point(x,y));
```

```
//переместить шарик из начальной клетки в конечную:
if form1.sbtPath.down= true then
  for i:= 0 to masPath[x, y]-1 do
  begin
    //координаты очередной клетки:
    p:= path[i].x; q:= path[i].y;
    //переместить шарик из очередной клетки в следующую -
    //цвет шарика:
    bc:= Cell[p, q].BallColor;
    //очередная клетка становится пустой и серой:
    Cell[p, q].BallColor:= bcNone;

    //исчезновение шарика:
    dr:= form1.dgPole.CellRect(p, q);
    index:= (ord(bc)-1) * period;
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+2);
    sleep(DelayMove);
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+1);
    sleep(DelayMove);
    //пустая клетка:
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top, 0);
    sleep(DelayMove);

    //в следующей появляется шарик:
    p:= path[i+1].x; q:= path[i+1].y;
    Cell[p, q].BallColor := bc;

    //появление шарика:
    dr:= form1.dgPole.CellRect(p, q);
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+1);
    sleep(DelayMove);
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+2);
    sleep(DelayMove);
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+3);
    sleep(160);
    Sound('move1');
  end
  else begin
    //координаты первой клетки:
    i:= path[0].x; j:= path[0].y;
    //цвет шарика:
    bc:= Cell[i, j].BallColor;
    //первая клетка становится пустой и серой:
    Cell[i, j].BallColor:= bcNone;
    form1.dgPoleDrawCell(Self, i, j, form1.dgPole.CellRect(i,
j),[]);
    //координаты последней клетки:
```

```
    p:= path[masPath[x, y]].x; q:= path[masPath[x, y]].y;
    //в ней появляется шарик:
    Cell[p, q].BallColor := bc;
    form1.dgPoleDrawCell(Self, p, q, form1.dgPole.CellRect(p,
q),[]);
    Sound('move1');
    sleep(160);
  end;
end;
```

Сначала мы возвращаем клетки поля в нормальное состояние (серый цвет), затем формируем путь шарика в массиве *path*:


```
//СФОРМИРОВАТЬ  ПУТЬ
procedure TPole.CreatePath(EndCell: TPoint);
var
  n, LenPath: integer;
  i, j, p, q: integer;

  //проверить координаты:
  function TestCoord(x,y: integer): boolean;
  begin
    Result:= true;
    if (x<0) or (x> width-1) or (y<0) or (y> height-1)or
       (masPath[x,y]<> n-1) then
      Result:= false;
  end;

begin
  //длина пути равна числу в конечной клетке:
  LenPath:= masPath[EndCell.x, EndCell.y];
  n:= LenPath;
  //конечная клетка пути:
  path[n]:= EndCell;
  //двигаемся от неё к начальной клетке:
  repeat
    //найти соседнюю клетку с числом n-1:
    p:= path[n].x;  q:= path[n].y;
    //проверяем соседние клетки:
    for i:= p - 1 to p + 1 do
      for j:= q - 1 to q + 1 do
        //нашли подходящую клетку:
        if ((i=p) or (j=q)) and TestCoord(i,j) then
        begin
          //записываем её координаты:
          path[n-1]:= Point(i,j);
          break;
        end;
    //ищем клетку с предыдущим номером:
    dec (n);
  until n<0;
end;
```

Шарик анимированно исчезает в начальной клетке и появляется в соседней, и так - пока не допрыгает до *конечной* клетки пути. Процесс перемещения шарика, сопровождаемый бульканьем и анимацией, может быстро надоесть, поэтому в программе должна быть возможность отключить последовательное перемещение шарика и сразу «телепортировать» его в заданную точку игрового пространства. Для этого мы установим *кнопку sbtPath* (**Hint**=Путь вкл-откл **GroupIndex**=2), действующую подобно *sbtSound*. Когда она *нажата* - , - будет показан весь путь шарика по клеткам, если *отжата* - , - он *сразу* появится в конечной клетке.

Если в результате хода образовалась фигура (линия или блок) достаточного размера, то её нужно уничтожить:

```
  //увеличить число сделанных ходов:
  inc(Hod.value);
  //и показать на экране:
  Hod.ShowValue;

  //если образовалась нужная фигура, уничтожить её:
  n:= DeleteFigs;
```

Поиском и ликвидацией фигур у нас занимается *функция tPole.DeleteFigs:*

```
//УНИЧТОЖИТЬ ФИГУРУ
function tPole.DeleteFigs: integer;
//возвращает число уничтоженных шариков
var
  i,j,n, index: integer;
  bc: TBallColor;
  dr: TRect;

  //найти на поле блоки нужной величины
  function GetBlocks(x,y: integer; Select: boolean): boolean;
  //x,y - координаты первой клетки
  var
    masPath2: TField;
    //список координат:
    CoordList: array[1..MAX_WIDTH * MAX_HEIGHT] of TPoint;
    //указатели в списке:
    ptrWrite, ptrRead: integer;
    p, q: integer;
    i, j: integer;

    //проверить координаты
    function TestCoord(x,y: integer): boolean;
    begin
```

```
      Result:= true;
      if (x<0) or (x> Width-1) or (y<0) or (y> Height-1)or
       (masPath2[x,y]<> GO) then Result:= false;
    end;

  begin
    //сначала в блоке один шарик:
    n:= 1;
    //отметить в массиве первую клетку:
    if Select then masPath[x,y]:= 1;

    //формируем массив лабиринта:
    for j:= 0 to height-1 do
      for i:= 0 to width-1 do
        //если в клетке есть шарик, то клетка непроходима:
        if Pole.Cell[i,j].BallColor<> bc then
          masPath2[i,j]:= STOP
        //иначе - клетка непроходима:
        else
          masPath2[i,j]:= GO;

    //заносим в список координаты начальной клетки:
    CoordList[1]:= Point(x,y);
    //в начальную клетку в массиве CellPath ставим BEG_CELL= 0:
    masPath2[x,y]:= 0;
    //устанавливаем указатель для считывания координат на начало
списка:
    ptrRead:= 1;
    //устанавливаем указатель для записи новых координат на
следующий индекс:
    ptrWrite:= 2;

    //двигаемся от начала списка к его концу, пока он не кончится:
    while ptrRead < ptrWrite do begin
      //координаты текущей клетки:
      p:= CoordList[ptrRead].x; q:= CoordList[ptrRead].y;
      //проверяем соседние с ней клетки:
      for i:= p - 1 to p + 1 do
        for j:= q - 1 to q + 1 do
          //если нашли соседнюю проходимую клетку,
          if ((i=p) or (j=q)) and TestCoord(i,j) then
          begin
            inc(n);
            //то записываем в неё 1:
            masPath2[i,j]:= 1;
            if Select then masPath[i,j]:= 1;
            //записываем координаты соседней клетки в конец
списка:
            CoordList[ptrWrite]:= Point(i,j);
            //перемещаем указатель:
            inc(ptrWrite);
          end;
      //переходим к следующей клетке в списке:
```

```
      inc(ptrRead);
  end;

    //найдена ли цепочка нужной длины?
    if n>= LenBlocks then Result:= true
    else Result:= false;
  end;

  //уничтожить блок
  procedure DeleteBlocks;
  var
    i,j: integer;
  begin
  //проверить клетки поля:
  for j:= 0 to height do
    for i:= 0 to width do begin
      //цвет очередного шарика:
      bc:= Cell[i,j].BallColor;
      //пустая или проверенная клетка:
      if (bc= bcNone) or (masPath[i,j]= 1) then Continue;
      //искать вправо:
      if GetBlocks(i,j,false) then GetBlocks(i,j,true);
    end;
  end;

  //найти на поле цепочки нужной длины
  function GetLenChain(x,y,dx,dy: integer; Select: boolean):
boolean;
  begin
    //сначала в цепочке один шарик:
    n:= 1;
    //отметить в массиве первую клетку:
    if Select then masPath[x,y]:= 1;
    //координаты следующей клетки:
    x:= x+dx; y:= y+dy;
    while (x>=0) and (x< width) and (y>=0) and (y< height) and
          (Cell[x,y].BallColor= bc) do begin
      //цепочка удлинилась:
      inc(n);
      //отметить клетку:
      if Select then masPath[x,y]:= 1;
      //координаты следующей клетки:
      x:= x+dx; y:= y+dy;
    end;
    //найдена ли цепочка нужной длины?
    if n>= LenLines then Result:= true
    else Result:= false;
  end;

  //уничтожить линии
  procedure DeleteLines;
  var i, j: integer;
  begin
```

```
  //проверить клетки поля:
  for j:= 0 to height do
    for i:= 0 to width do begin
      //цвет очередного шарика:
      bc:= Cell[i,j].BallColor;
      if bc= bcNone then Continue; // - пустая клетка
      //искать вправо:
      if GetLenChain(i,j,1,0,false) then
GetLenChain(i,j,1,0,true);
      //искать вниз:
      if GetLenChain(i,j,0,1,false) then
GetLenChain(i,j,0,1,true);
      //искать вправо-вниз:
      if GetLenChain(i,j,1,1,false) then
GetLenChain(i,j,1,1,true);
      //искать влево-вниз:
      if GetLenChain(i,j,-1,1,false) then GetLenChain(i,j,-
1,1,true);
    end;
  end;


begin
  //обнулить массив masPath:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      masPath[i,j]:= 0;

     if TypeFig= tfBlocks then DeleteBlocks
     else Deletelines;

  n:= 0;
  //подсчитать число уничтожаемых шариков:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      if masPath[i,j]= 1 then inc(n);
  Result:= n;
  if n= 0 then exit;

  //уничтожить шарики:
  for n:= 1 to 5 do begin
    for j:= 0 to height-1 do
      for i:= 0 to width-1 do
        if masPath[i,j]= 1 then begin
          //исчезновение шарика:
          dr:= form1.dgPole.CellRect(i, j);
          bc:= Cell[i,j].BallColor;
          index:= (ord(bc)-1) * period+n+5;
          if n=5 then begin
            index:= 0; // - пустая клетка
            Cell[i,j].BallColor:= bcNone;
          end;
```

```
          form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index);
        end;
    sleep(DelayDestroy);
  end;
  Sound('destroy2');
end;
```

Так как линии и блоки отличаются по форме, то ими занимаются *разные* процедуры:


```
     if TypeFig= tfBlocks then DeleteBlocks
     else Deletelines;
```

Длину *линий* мы вычисляем так. Находим на поле шарик и считаем, сколько он имеет соседей справа, снизу, а также по диагоналям вниз-вправо и вниз-влево. Если удалось найти цепочки нужной длины, фукция *GetLenChain* возвращает *TRUE*, и вызывается вторично, чтобы отметить шарики в длинных линиях: *masPath[]:= 1*.

Число шариков в *блоке* мы подсчитываем по тому же алгоритму, по которому отыскивали путь в лабиринте и окрашивали клетки поля. После применения алгоритма в массиве *masPath* единицами будут отмечены клетки, образующие блоки.

Таким образом, независимо от типа собираемых фигур, по содержимому массива *masPath* мы легко определим, какие шарики нужно уничтожить и сколько их:

```
  n:= 0;
  //подсчитать число уничтожаемых шариков:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      if masPath[i,j]= 1 then inc(n);
  Result:= n;
  if n= 0 then exit;
```

Если ни одной фигуры не найдено (*n= 0*), то и уничтожать нечего. В противном случае «фигуристые» шарики уничтожаются – также анимированно (5 кадров).

Функция *DeleteFigs* возвращает число шариков в фигуре (оно может быть равно *нулю*, если в результате хода не возникло ни одной фигуры), и мы можем вывести новую *статистику* игры:

```
  //вычислить и показать набранные очки:
  Score.SetValue(n);
```

```
Score.ShowValue;

//вычислить и показать игровой тонус:
Tonus.GetValue;
Tonus.ShowValue;

//если был сделан результативный ход и на поле есть шарики,
//то ждать следующего хода:
if (n> 0) and (Tonus.EmptyPole= false) then exit;
```

Также мы должны учесть, что по правилам игры после результативного хода (*n> 0*) игрок может сделать ещё один ход – если, конечно, на поле остались пустые клетки:


```
//если для новых шариков не осталось места, то игра проиграна:
if Tonus.EmptyCell < NumBalls.nPotom then begin
  GameOver;
  exit
end;
```

```
//если поле пустое - победа:
if Tonus.EmptyPole= true then begin
  GameWin;
  exit
end;
```

Если игру можно продолжить, то мы выводим на поле новую порцию шариков:

```
//найти место для новых шариков:
n:= 0;
repeat
    i:= random(Width);
    j:= random(Height);
    if Cell[i, j].BallColor= bcNone then begin
      //поместить в пустую клетку «следующий» шарик:
      Cell[i, j].BallColor:= NextBalls[n+1];
      Cell[i, j].new:= true;
      inc(n)
    end;
until n= NumBalls.nPotom;
//появление шариков с бульканием:
CreateBalls;
```

Если и эти шарики образовали с уже имеющимися на поле новые фигуры, их также следует уничтожить:

```
//если образовалась нужная фигура, уничтожить её:
n:= DeleteFigs;
```

```
  //вычислить и показать набранные очки:
  Score.SetValue(n);
  Score.ShowValue;

  //вычислить и показать игровой тонус:
  Tonus.GetValue;
  Tonus.ShowValue;
```

В сетке-табло показываем следующие шарики:

```
  //показать следующие шарики:
  InitNextBalls;
  ShowNextBalls;
```


После самоуничтожения фигур поле могло очиститься полностью:

```
  //если поле пустое - победа:
  if Tonus.EmptyPole= true then begin
    GameWin;
    exit
  end;
```

Но могла возникнуть и такая позиция, в которой ходов у игрока нет:

```
  //если ходов нет - поражение:
  if Tonus.FullPole= true then begin
    GameOver;
    exit
  end;
```

Обычно игра рано или поздно заканчивается печально:

```
//ИГРА ЗАКОНЧЕНА
procedure tPole.GameOver;
begin
  form1.imgPor.visible:= true;
  sleep(1000);
  application.ProcessMessages;
  sound('gameover');
  sleep(7500);
  form1.imgPor.visible:= false;
  sleep(1000);
  application.ProcessMessages;
  Score.SetRec;
  form1.NewGame;
end;
```

Это событие сопровождается противной музыкой и обидной надписью (Рис. 5.33) (картинка для неё находится в компоненте *imgPor*).



Рис. 5.33. Финита ла комедия!

После чего программа определяет, не стали ли вы новым рекордсменом, и начинает новую игру.

Очень редко кому доведётся на своём веку пережить более счастивый финал (Рис. 5.34).

```
//ПОБЕДА!
procedure tPole.GameWin;
begin
  form1.imgPobeda.visible:= true;
  sleep(1000);
  application.ProcessMessages;
  sound('pobeda');
  sleep(7500);
  form1.imgPobeda. visible:= false;
  Score.SetRec;
  form1.NewGame;
end;
```



Рис. 5.34. Бывает всякое – это жизнь!

«Победная» картинка загружена в компонент *imgPobeda*.

Эти картинки хранятся в *папке result*.



Исходный код программы находится в папке **Линии**.

«Требую продолжения банкета!», или Где начало того конца, которым оканчивается начало?

Король умер. Да здравствует король!

Круговращение французких королей

Собирание шариков в линии и блоки – процесс сугубо абстрактный, весьма далёкий от насущных потребностей человека, поэтому мы попытаемся придать ему некий рациональный смысл. Например, уничтожение групп шариков по достижении ими «критической массы» можно легко интерпретировать как взаимодействие атомов. Собирание атомов в молекулы - это модные сейчас нано-технологии, то есть процессы, вполне достойные просвещённого ума! Эти примеры трансформирования игры можно продолжать сколь угодно долго, поэтому мы переходим от залихватских фантазий к вполне практическому программированию учебно-воспитательных приложений.

Молекулярный конструктор

Не робей – собирай скорей!

Антинаучный призыв

В *первой* игре мы будем собирать *молекулы* из атомов. Понятно, что между молекулами и блоками шариков принципиальной разницы нет (в смысле игры!), поэтому исправления исходной программы будут минимальны.

В первую очередь, наш конструктор предназначен для школьников, здесь важен сам по себе процесс сборки молекул из атомов, так что все лишние настройки программы вместе с кнопками мы безжалостно удалим, дабы они не отвлекали игрока от сущности бытия, и оставим только самое необходимое для имитации сборки молекул. Прочтите *справочный файл* к программе, из которого легко определить, что именно нам нужно изменить в программе *Линии*:

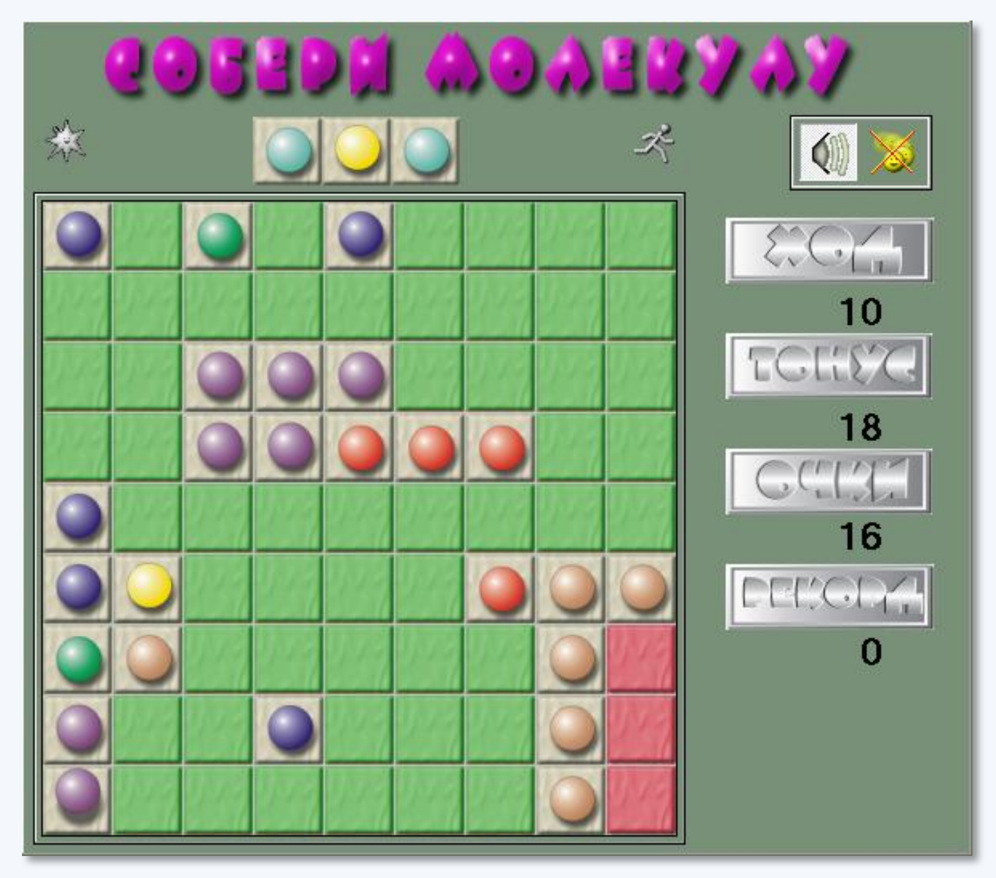
Ваша задача – собирать молекулы, состоящие не менее чем из *шести* атомов одного вида. Атомы в молекуле могут соединяться только по горизонтали и вертикали, но не по диагонали.

Всего имеется 7 разновидностей атомов, представленных в игре шариками разного цвета.

Как только вы соберёте молекулу нужной величины, в ней произойдёт цепная реакция и она исчезнет, а вы за этот научно-технический подвиг получите призовые очки и сможете продолжить сборку (сделать ещё один ход). Чем длиннее получится цепочка, тем больше очков вы за неё получите. В противном случае появятся ещё три новых атома.

Чтобы переместить атом с одного места на другое, щёлкните его мышкой. Он перейдёт в возбуждённое состояние и будет подпрыгивать. Затем щёлкните на той клетке игрового поля, куда бы вы хотели переместить атом. Но имейте в виду, что атом нельзя переместить на уже занятую клетку или на клетку *красного* цвета. Если вы выбрали какой-то атом, а затем передумали, то просто щёлкните на другом атоме – и он станет выбранным.

Число оставшихся у вас ходов (при условии, что вы не сможете больше собрать ни одну молекулу) указывается справа, под надписью *Тонус*. Там же вы найдёте и другую полезную информацию:



Молекулярный конструктор в действии

Игра заканчивается, если на поле не останется места для новых атомов (*проигрыш*) или все атомы будут уничтожены (*выигрыш*).

В любое время вы можете *возобновить* игру, нажав кнопку , а также *закончить* игру, нажав кнопку . Между этими кнопками показана следующая порция атомов.

Вы также можете отключить *звук* с помощью кнопки и показ пути с помощью кнопки .

Таким образом, нужно исправить только 1 строчку кода:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
…
  TypeFig:=tfBlocks;
end;
```

Заменить картинки в компоненте *imlPic* (Рис. 5.35).

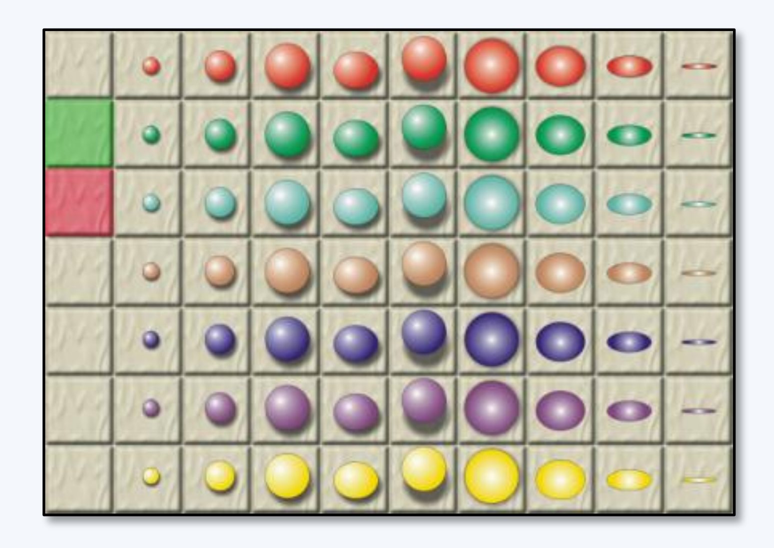


Рис. 5.35. Атомы во всей красе!

И, конечно, заголовок окна приложения (Рис. 5.36).



Рис. 5.36. Название игры



Исходный код программы находится в папке **MakeMol**.

Экологический субботник

Чистота – залог здоровья.

Такая пословица

Чисто не там, где убирают,
а там, где не сорят.

Тоже

Не надо мусорить, да?!

Гринписовский призыв

Если в первой игре мы воспользовались таким свойством игры *Линии* как *собирание* фигур, то во *второй* нам пригодится другое её свойство – мы будем *уничтожать* фигуры, освобождая поле от шариков - так мы будем наводить чистоту в родном городе.

В соответствии с новыми задачами изменим интерфейс программы. Заголовок окна (Рис. 5.37).



Рис. 5.37. Название игры

Таблички с надписями (Рис. 5.38).







Рис. 5.38. Статистические таблички

Вместо таблички *Тонус* появятся 2 новые (Рис. 5.39).





Рис. 5.39. Специфические таблички

Шарики заменит городской *мусор* – огрызок яблока, шляпа, опавшие листья и старый башмак (Рис. 5.40).

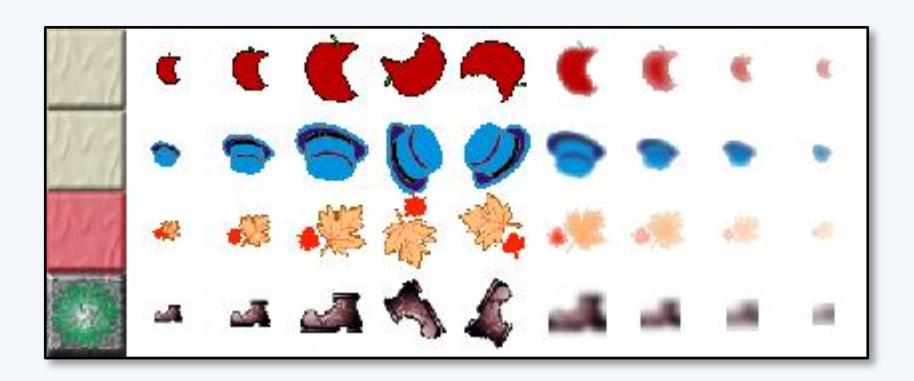


Рис. 5.40. Типичная картина!

В игре используются только *три* первых артефакта, иначе выиграть будет затруднительно!

Новые условия игры мы учитываем при создании формы. Сразу после запуска приложения в городе появляются 10 «штук» мусора, а затем – по 3. Разновидностей мусора, как мы только что выяснили, тоже 3. Собирать мусор следует в кучи, имеющие форму *блоков*:

```
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
begin
…
  //шарики:
  NumBalls.nSrazu:= 10;
  NumBalls.nPotom:= 3;

  //количество цветов:
  NumBalls.nColor:= 3;

  TypeFig:=tfBlocks;
…
end;
```

Поскольку величина поля не изменяется и всегда равна 9 х 9 клеток, то и размер клеток можно *увеличить*, чтобы картинка лучше «смотрелась»:

```
var
  //размеры клетки сетки:
  wCell: integer = 43;
  hCell: integer = 43;
```

Соответственно придётся исправить и *класс tTonus*:

```
  //игровой тонус:
  type tTonus= class
    //число оставшихся ходов:
    value: integer;
    //число пустых клеток на поле:
    EmptyCell: integer;
    //есть ли на поле шарики?
    EmptyPole: boolean;
    //есть ли на поле хотя бы одна свободная клетка?
    FullPole: boolean;
    //процент очищенной площади города:
    Chisto: integer;
    //число клеток с мусором:
    Musor: integer;
    constructor Create;
    procedure ShowValue;
    procedure GetValue;
  end;
```



```
///////////////////////////////////////////////////////////
///////////////////  ИГРОВОЙ ТОНУС ////////////////////////
///////////////////////////////////////////////////////////

constructor tTonus.Create;
begin
  inherited;
  Chisto:= 0;
end;

//ВЫЧИСЛИТЬ ПЛОЩАДЬ
procedure tTonus.GetValue;
var
  i, j: integer;
begin
  //подсчитать число пустых клеток на поле:
  EmptyCell:= 0;
  for j:= 0 to Pole.Height-1 do
    for i:= 0 to Pole.Width-1 do
      if Pole.Cell[i, j].BallColor= bcNone then inc(EmptyCell);

  chisto:= 0;
  for j:= 0 to Pole.Height-1 do
    for i:= 0 to Pole.Width-1 do
      if Pole.masClear[i, j]= true then inc(Chisto);

  value:= Chisto*100 div (Pole.Height*Pole.Width);

  //мусор:
  musor:= Pole.Height * Pole.Width - EmptyCell;
  //проверить, не пусто ли поле:
  if EmptyCell = Pole.Height * Pole.Width then
    EmptyPole:= true
```

```
  else
    EmptyPole:= false;
  //проверить, есть ли на поле свободные клетки:
  if EmptyCell = 0 then
    FullPole:= true
  else
    FullPole:= false;
end;

//ВЫВЕСТИ ЗНАЧЕНИЕ ТОНУСА НА ЭКРАН
procedure tTonus.ShowValue;
begin
  form1.lblTonus.Caption:= inttostr(value)+'%';
  form1.lblTonus.Invalidate;
  form1.lblMusor.Caption:= inttostr(Musor);
  form1.lblMusor.Invalidate;
  application.ProcessMessages
end;
```


В *классе tScore* мы изменим подсчёт набранных очков:

```
/////////////////////////////////////////////////////////////
///////////////////////// ОЧКИ //////////////////////////////
/////////////////////////////////////////////////////////////

…

//ПОДСЧИТАТЬ НАБРАННЫЕ ОЧКИ
procedure tScore.SetValue(n: integer);
begin
  if hod.value > 80 then dec(value)
  else
    if n> 0 then value:= value + 4*n - 9;

end;

…
```

Фоном для поля послужат 2 фотографии города – чёрно-белая (метафора *грязного* города) и цветная (*чистого*) (Рис. 5.41).





Рис. 5.41. Наглядный призыв к чистоте!

Их размеры для поля 9 х 9 клеток и размера клеток 43 х 43 пикселя должны быть 387 х 387 пикселей. Хранить картинки разумно в компоненте *imlGorod: TImageList* (5.42).

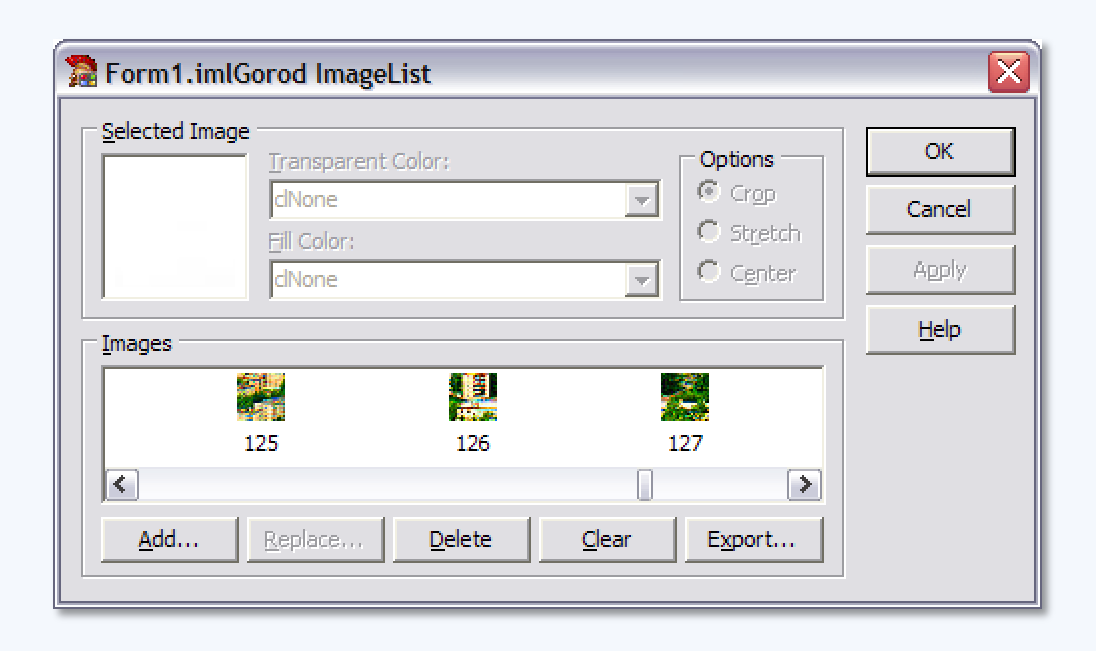


Рис. 5.42. Городской фотоальбом

Всего из двух фотографий получится 9 x 9 x 2 = 162 маленьких квадратика. Естественно, фотографии не нужно нарезать вручную – при загрузке их в компонент *TImageList* эта операция выполняется автоматически!

Так как теперь каждая клетка поля имеет *фон* (изменяющийся по ходу игры), на который накладываются картинки с мусором, то процедура отрисовки клеток поля становится более сложной:

```
//ОТРИСОВАТЬ СЕТКУ
procedure TForm1.dgPoleDrawCell(Sender: TObject; ACol, ARow:
Integer;
  Rect: TRect; State: TGridDrawState);
var
  dr: TRECT;
  index: integer;
  bmp: TBitmap;
begin
  //размеры картинок:
  dr:= Bounds(Rect.Left, Rect.Top, wCell, hCell);

  //создать промежуточный растр:
  bmp:= TBitmap.Create;
  bmp.Width:= wCell;
  bmp.Height:= hCell;

  //картинка с шариком в ячейке:
  index:= (ord(Pole.Cell[ACol, ARow].BallColor));

  if index> 0 then begin//- в клетке есть "шарик"
    index:= GetIndex(ACol, ARow);
    //скопировать картинку с городом в ячейку:
```

```
    imlGorod.Draw(bmp.Canvas, 0, 0, index);

    //затем - картинку с "шариком":
    index:= (ord(Pole.Cell[ACol, ARow].BallColor));
    index:= (index-1)* period + 3;
    imlPic.Draw(bmp.Canvas, 0, 0, index);
    dgPole.canvas.Draw(Rect.Left, Rect.Top, bmp);
    end
  else begin              //- пустая клетка
    index:= GetIndex(ACol, ARow);
    //скопировать картинку с городом в ячейку:
    imlGorod.Draw(dgPole.Canvas, dr.left, dr.top, index);
    //отметить доступные и недоступные клетки:
    if Pole.Cell[ACol, ARow].CellColor<> ccGray then begin
      //нарисовать контурный квадрат:
      InflateRect(dr,-3,-3);
      dgPole.Canvas.Brush.Style:= bsClear;
      dgPole.Canvas.Pen.Width:= 1;
      if Pole.Cell[ACol, ARow].CellColor= ccGreen then
        //цвет линий:
         dgPole.Canvas.Pen.Color:= rgb(0,255,0)
      else
         dgPole.Canvas.Pen.Color:= rgb(255,0,0);
      dgPole.Canvas.Rectangle(dr.left, dr.top, dr.right,
dr.bottom);
    end;
  end;
  bmp.Free;
end;
```

Сначала мы создаём в памяти компьютера промежуточный растр *bmp*, затем копируем в него картинку с городом. Тут нужно уметь различать грязные и чистые участки поля:

```
function TForm1.GetIndex(X, Y: integer): integer;
begin
  if not Pole.masClear[x,y] then  //gorod2
    result:= y*9+ x
  else
    result:= y*9+ x+81;
end;
```

Индексы картинок *грязного* поля в *imlGorod* равны 0..80, *чистого* - 81..161.

Если в клетке есть мусор, мы копируем соответствующую картинку поверх фона. Если клетка без мусора, то выводим в неё рамку *зелёного* (доступная клетка) или *красного* (недоступная) цвета (когда выделен какой-либо «шарик» на поле, естественно!). Скомбинированную картинку с фоном и мусором рисуем в ячейке поля.

Теперь рассмотрим изменения в классе *tPole*. Прежде всего, нужно изменить значение переменной *LenBlocks*, так как минимальный размер блока (кучи) равен трём (иначе «победить» мусор будет очень трудно):

```
//////////////////////////////////////////////////////////
///////////////////////// ПОЛЕ ///////////////////////////////
//////////////////////////////////////////////////////////
constructor tPole.Create;
begin
  inherited create;
  LenBlocks:= 3;
  DelayDestroy:= 150;
  DelayCreate:= 80;
  DelayMove:= 40;
end;
```


Подпрыгивающий шарик в этой игре будет крутящимся в вихре мусором:

```
//ПОДПРЫГИВАЮЩИЙ ШАРИК
procedure TPole.Animation;
//x. y - координаты клетки поля;
//index - номер картинки;
var
  dr: Trect;
  x,y, index: integer;

begin
  //все повторения выполнены:
  if (AnimateCell.Rep<> 0) and (AnimateCell.Exec= AnimateCell.Rep)
then
  begin
    form1.timer2.Enabled:= false;
    exit;
  end;
  form1.timer2.Interval:= AnimateCell.Interval;
  x:= AnimateCell.x;
  y:= AnimateCell.y;
  //квадрат клетки:
  dr:= form1.dgPole.CellRect(x,y);
  //город:
  index:= form1.GetIndex(x, y);
  //скопировать картинку с городом в ячейку:
  form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index);

  //номер текущей картинки:
  index:= AnimateCell.Index[AnimateCell.ptrIndex];
  //вывести текущую картинку:

  form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top, Index);
  if (AnimateCell.sound<> '') and
```

```
    (AnimateCell.SoundIndex= AnimateCell.ptrIndex)
  then
    Sound(AnimateCell.sound);

  inc(AnimateCell.ptrIndex);
  if AnimateCell.ptrIndex> AnimateCell.Frame then
    AnimateCell.ptrIndex:= 1;

  if AnimateCell.rep<> 0 then
    inc(AnimateCell.exec);
end;
```


Изменения в этой и следующих процедурах обусловлены необходимостью подкладывать фон под «шарик»:

```
//ПЕРЕМЕСТИТЬ ШАРИК
procedure tPole.DoMoveBall(X, Y: Integer);
//X, Y - координаты клетки
var
  dr: TRECT;
  index, i,j: integer;
  bc: TBallColor;
  p, q: integer;
  ind: integer;
begin
  //убрать выделение с клеток:
  DeactivateBall;
  //создать путь:
  CreatePath(Point(x,y));
  //переместить шарик из начальной клетки в конечную:

if form1.sbtPath.down= true then  //показать путь
  for i:= 0 to masPath[x, y]-1 do
  begin
    //координаты очередной клетки:
    p:= path[i].x; q:= path[i].y;
    //переместить шарик из очередной клетки в следующую -
    //цвет шарика:
    bc:= Cell[p, q].BallColor;
    //очередная клетка становится пустой и серой:
    Cell[p, q].BallColor:= bcNone;

    //исчезновение шарика:
    dr:= form1.dgPole.CellRect(p, q);
    index:= (ord(bc)-1) * period;
    //город:
    //скопировать картинку с городом в ячейку:
    ind:= form1.GetIndex(p, q);
    form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
ind);
    //пустая клетка:
```

```
    form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
Ind);

    sleep(DelayMove);

    //в следующей клетке появляется шарик:
    p:= path[i+1].x; q:= path[i+1].y;
    Cell[p, q].BallColor := bc;

    //появление шарика:
    dr:= form1.dgPole.CellRect(p, q);
    //скопировать картинку с городом в ячейку:
    form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
    form1.GetIndex(p, q));
    form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+3);
    sleep(DelayMove);
  end
  else begin
    //координаты первой клетки:
    i:= path[0].x; j:= path[0].y;
    //цвет шарика:
    bc:= Cell[i, j].BallColor;
    //первая клетка становится пустой и серой:
    Cell[i, j].BallColor:= bcNone;
    form1.dgPoleDrawCell(Self, i, j, form1.dgPole.CellRect(i,
j),[]);
    //координаты последней клетки:
    p:= path[masPath[x, y]].x; q:= path[masPath[x, y]].y;
    //в ней появляется шарик:
    Cell[p, q].BallColor:= bc;
    form1.dgPoleDrawCell(Self, p, q, form1.dgPole.CellRect(p,
q),[]);
    Sound('move1');
    sleep(160);
  end;
end;
```

```
//УНИЧТОЖИТЬ ФИГУРУ
function tPole.DeleteFigs: integer;
//возвращает число уничтоженных шариков
var
  i,j,n, index: integer;
  bc: TBallColor;
  dr: TRect;

  //найти на поле блоки нужной величины
  function GetBlocks(x,y: integer; Select: boolean): boolean;
  //x,y - координаты первой клетки
  var
    masPath2: TField;
    //список координат:
```

```
    CoordList: array[1..MAX_WIDTH * MAX_HEIGHT] of TPoint;
    //указатели в списке:
    ptrWrite, ptrRead: integer;
    p, q: integer;
    i, j: integer;

    //проверить координаты
    function TestCoord(x,y: integer): boolean;
    begin
      Result:= true;
      if (x<0) or (x> Width-1) or (y<0) or (y> Height-1)or
       (masPath2[x,y]<> GO) then Result:= false;
    end;

  begin
    //сначала в блоке один шарик:
    n:= 1;
    //отметить в массиве первую клетку:
    if Select then masPath[x,y]:= 1;

    //формируем массив лабиринта:
    for j:= 0 to height-1 do
      for i:= 0 to width-1 do
        //если в клетке есть шарик, то клетка непроходима:
        if Pole.Cell[i,j].BallColor<> bc then
          masPath2[i,j]:= STOP
        //иначе - клетка непроходима:
        else
          masPath2[i,j]:= GO;

    //заносим в список координаты начальной клетки:
    CoordList[1]:= Point(x,y);
    //в начальную клетку в массиве CellPath ставим BEG_CELL= 0:
    masPath2[x,y]:= 0;
    //устанавливаем указатель для считывания координат на начало
списка:
    ptrRead:= 1;
    //устанавливаем указатель для записи новых координат на
следующий индекс:
    ptrWrite:= 2;

    //двигаемся от начала списка к его концу, пока он не кончится:
    while ptrRead < ptrWrite do begin
      //координаты текущей клетки:
      p:= CoordList[ptrRead].x; q:= CoordList[ptrRead].y;
      //проверяем соседние с ней клетки:
      for i:= p - 1 to p + 1 do
        for j:= q - 1 to q + 1 do
          //если нашли соседнюю проходимую клетку,
          if ((i=p) or (j=q)) and TestCoord(i,j) then
          begin
            inc(n);
            //то записываем в неё 1:
```

```
            masPath2[i,j]:= 1;
            if Select then masPath[i,j]:= 1;
            //записываем координаты соседней клетки в конец
списка:
            CoordList[ptrWrite]:= Point(i,j);
            //перемещаем указатель:
            inc(ptrWrite);
          end;
      //переходим к следующей клетке в списке:
      inc(ptrRead);
  end;

    //найдена ли цепочка нужной длины?
    if n>= LenBlocks then Result:= true
    else Result:= false;
  end;

  //уничтожить блок
  procedure DeleteBlocks;
  var
    i,j: integer;
  begin
  //проверить клетки поля:
  for j:= 0 to height do
    for i:= 0 to width do begin
      //цвет очередного шарика:
      bc:= Cell[i,j].BallColor;
      //пустая или проверенная клетка:
      if (bc= bcNone) or (masPath[i,j]= 1) then Continue;
      //искать вправо:
      if GetBlocks(i,j,false) then GetBlocks(i,j,true);
    end;
  end;

begin
  //обнулить массив masPath:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      masPath[i,j]:= 0;

  DeleteBlocks;

  n:= 0;
  //подсчитать число уничтожаемых шариков:
  for j:= 0 to height-1 do
    for i:= 0 to width-1 do
      if masPath[i,j]= 1 then inc(n);
  Result:= n;
  if n= 0 then exit;

  //уничтожить шарики:
  for n:= 1 to 5 do begin
    for j:= 0 to height-1 do
```

```
      for i:= 0 to width-1 do
        if masPath[i,j]= 1 then begin
          masClear[i,j]:= true;
          //исчезновение шарика:
          dr:= form1.dgPole.CellRect(i, j);
          bc:= Cell[i,j].BallColor;
          index:= (ord(bc)-1) * period+n+5;
          if n=5 then begin
            index:= form1.GetIndex(i,j); // - пустая клетка
            form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left,
dr.top, index);
            Cell[i,j].BallColor:= bcNone;
          end
          else
            //скопировать картинку с городом в ячейку:
            form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left,
dr.top, form1.GetIndex(i, j));
            form1.imlPic.Draw(form1.dgPole.Canvas, dr.left,
dr.top, index);
        end;
    sleep(DelayDestroy);
  end;
  Sound('destroy2');
end;
```



```
//ПОЯВЛЕНИЕ ШАРИКОВ:
procedure TPole.CreateBalls;
var
  i, j, index, ind: integer;
  dr: TRECT;
begin
  //r:= form1.GetRect;
  for j:= 0 to Pole.Height-1 do
    for i:= 0 to Pole.Width-1 do begin
      dr:= form1.dgPole.CellRect(i, j);
      index:= (ord(Pole.Cell[i, j].BallColor)-1) * period;
      if Pole.Cell[i, j].new= true then begin
        Pole.Cell[i, j].new:= false;

        //скопировать картинку с городом в ячейку:
        ind:= form1.GetIndex(i, j);
        form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
ind);

        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+1);
        Delay(DelayCreate);
        form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
ind);
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+2);
        Delay(DelayCreate);
```

```
        form1.imlGorod.Draw(form1.dgPole.Canvas, dr.left, dr.top,
ind);
        form1.imlPic.Draw(form1.dgPole.Canvas, dr.left, dr.top,
index+ 3);
        Sound('buljk');
      end;
    end;
end;
```

Необходимо исправить и *процедуру DoHod*, так как изменились правила игры:


```
//ПЕРЕМЕСТИТЬ ШАРИК
procedure tPole.DoHod(X, Y: Integer);
//X, Y - координаты клетки
var
  n, i, j: integer;
begin
…

  //если был сделан результативный ход и на поле есть шарики,
  //то ждать следующего хода:
  if (n> 0) and (Tonus.EmptyPole= false) then exit;

  //вывести следующие шарики NextBalls -

  //если для новых шариков не осталось места, то игра проиграна:
  if Tonus.EmptyCell < NumBalls.nPotom then begin
    GameOver;
    exit
  end;

  //если поле пустое - победа:
  if (Tonus.Chisto= Pole.width*Pole.height) and (Tonus.Musor= 0)
then begin
    GameWin;
    exit
  end;

  //найти место для новых шариков:
  n:= 0;
  repeat
    i:= random(Pole.Width);
    j:= random(Pole.Height);
    if Pole.Cell[i, j].BallColor= bcNone then begin
      //поместить в пустую клетку шарик случайного цвета:
      Pole.Cell[i, j].BallColor:= NextBalls[n+1];
      Pole.Cell[i, j].new:= true;
      inc(n)
    end;
  until n= NumBalls.nPotom;
```

```
…

  //если поле пустое - победа:
  if (Tonus.Chisto= Pole.width*Pole.height) and (Tonus.Musor= 0)
then begin
    GameWin;
    exit
  end;

  //если ходов нет - поражение:
  if Tonus.FullPole= true then begin
    GameOver;
    exit
  end;
end;
```

В этой процедуре новый мусор появляется в тех клетках, где его нет:

```
  //найти место для новых шариков:
  n:= 0;
  repeat
    i:= random(Pole.Width);
    j:= random(Pole.Height);
    if Pole.Cell[i, j].BallColor= bcNone then begin
      //поместить в пустую клетку шарик случайного цвета:
      Pole.Cell[i, j].BallColor:= NextBalls[n+1];
      Pole.Cell[i, j].new:= true;
      inc(n)
    end;
  until n= NumBalls.nPotom;
```

Это значит, что игроку придётся убирать мусор и с уже очищенных участков города. Если вам это кажется несправедливым, вы можете помещать новый мусор только в грязные клетки…

А наградой за победу в игре послужит новая картинка в компоненте *imgPobeda* (Рис. 5.43).



Рис. 5.43. Заработал – получи грамоту!

Справочный файл к программе:

Ваша задача в этой игре – очистить родной город от мусора. И тогда - сначала грязный и серый – он станет цветным и радостным!

Город разбит на 81 участок (это клеточки на игровом поле). На каждом из них может находиться мусор трёх разных видов, обозначенных огрызком яблока, рваной шляпой и опавшими листьями.

Мусор нужно собирать в кучки. Не менее трёх соседних по вертикали и горизонтали участков должны быть заняты мусором одного вида. Если вам удастся это сделать, то мусор исчезнет, город на этих участках станет чисто-цветным, а вы получите призовые очки. Чем больше размер кучки, тем больше очков за неё вы получите, так что не ленитесь, гребите шибче!

Чтобы переместить мусор с одного участка города на другой, щёлкните на участке с мусором, а затем на «целевом» участке. Но имейте в виду, что между этими двумя участками должен существовать путь, свободный от мусора. В игре все доступные участки выделяются *зелёной* рамкой, недоступные – *красной* (они появляются после выбора участка с мусором). Кроме того, вы не можете всё «валить в одну кучу» - на грязный участок мусор переносить нельзя. Если вы выбрали какой-то участок с мусором, а затем передумали, то просто щёлкните на другом участке – он станет выбранным.

Как только вы соберёте кучку мусора нужной величины, она исчезнет, а вы сможете продолжить свою работу (сделать ещё один ход). В противном случае «свежий» мусор появится ещё на трёх участках города.

Число грязных участков указывается справа, под надписью *Мусор*, процент убранной территории – под надписью *Чисто*:



Если вы не справитесь с задачей за 80 ходов, то дальше призовые очки вам начисляться не будут, а наоборот, вы будете терять по одному очку за каждый лишний ход.

Игра заканчивается, когда город будет полностью очищен от мусора (*Чисто* = 100%) и ни на одном участке не останется мусора (Мусор = 0).

В любое время вы можете начать новую игру, нажав кнопку , а также закончить игру, нажав кнопку . Между этими кнопками показан тот мусор, который благодарные горожане уже приготовили для вас. Он скоро появится, ждите!

Вы также можете отключить *звук* с помощью кнопки и *показ пути* с помощью кнопки .

Город ждёт – все на экологический субботник!

Твори, выдумывай, пробуй!

Правильный лозунг

На мой взгляд, этот вариант игры не только более «воспитательный», но и гораздо интереснее классического: сыграйте несколько «партий» - вам понравится! Подобным же образом вы можете наполнить новым смыслом и прочие известные игры, а приём с фоном в сетке пригодится вам в других проектах.



Исходный код программы находится в папке **Gorod**.

Факультатив 6. «Что наша «Жизнь»? – Игра!», или Клеточный автомат Джона Конвея

Игру *Жизнь* - самый известный в мире *клеточный автомат* – придумал в 1970 году математик из Кембриджского университета Джон Конвей. Как и все прочие клеточные автоматы, она моделирует развитие некой начальной популяции клеток в соответствии с немногочисленными законами (правилами) их существования.

Законы *Жизни*

Таковы суровые законы жизни.
Или, короче выражаясь,
жизнь диктует нам свои суровые законы.

И. Ильф, Е. Петров,
Золотой теленок

Мир, или вселенная, в которой живут организмы, представляет собой *бесконечную* плоскую сетку, разделённую на квадратики – ячейки, клетки. Мы (или наша программа) в качестве Творца засеваем некоторые ячейки организмами, которые образуют *начальное* (нулевое) поколение. Через некоторое время в популяции происходят такие изменения:

1. Каждая клетка (организм) имеет 8 *соседних* клеток по всем направлениям (вселенная бесконечна!). Если пустая ячейка имеет *трёх* живых соседей, то в ней возникает новая жизнь. Обозначим это правило зарождения жизни так: *B(orn)=3*.
2. Клетка *погибает*, если у неё слишком мало живых соседей (1) или слишком много (4,5,6,7,8).
3. Отсюда следует, что *продолжает жить* только клетка, имеющая 2-х или 3-х соседей. Это правило продолжения жизни записывается так: *S(urvive)=2,3*.

Тут следует заметить, что время во вселенной изменяется дискретно и каждому моменту времени соответствует своя популяция клеток. При формировании следующего поколения учитываются все клетки, уже имеющиеся на поле (живые на этот момент), но не те, которые возникают вновь (рождающиеся в этот момент).

Как видите, правила жизни в клеточном мире очень просты, что отнюдь не мешает достаточно примитивным начальным популяциям образовывать

очень сложные конфигурации клеток. Легко догадаться, что развитие жизни во вселенной определяется только количеством и взаимным расположением клеток в *исходном* состоянии. Найдено множество любопытных начальных структур клеток, но это не должно сдерживать ваши благородные порывы в создании новых миров!

Кроме того, изменение начальной конфигурации клеток существенно зависит от законов существования организмов. Описанные выше правила относятся к *классическому* варианту игры, придуманному Конвеем. Но поскольку каждая клетка может иметь от 0 до 8 соседей, то значения *B* и *S* также изменяются в этом диапазоне, причём *B* и *S* могут иметь и *несколько* значений, что даёт множество сочетаний этих параметров. Конечно, не все они дают интересные результаты, но, возможно, вы найдёте ошеломляющие примеры разнообразия жизни в разных вселенных!

Ну, вот и всё об истории *Жизни*, пора и за дело браться!

Как построить вселенную, или Даём жизни!

Жизнь - это сложная штука,
но эта сложная штука открывается просто,
как ящик.
Надо только уметь его открыть.
Кто не может открыть, тот пропадает.

И. Ильф, Е. Петров,
Двенадцать стульев

Сама вселенная с успехом может быть представлена нашим любимым компонентом *TDrawGrid*. Единственное, что омрачает радость, - *ограниченные* размеры компонента, а, значит, и нашей реализации вселенной. К сожалению, бесконечной сделать её не удастся в любом случае, поэтому протяжённые популяции клеток будут искажаться на границах мира. Избежать этого нельзя, но у нас имеется 2 варианта реакции на краевые эффекты: оставить мир *плоским* и игнорировать все клетки, уходящие за пределы компонента, либо свернуть плоскость в *сферу* и сделать мир «бесконечным», хотя и небольшим. В *первом* случае мы не только потеряем часть популяции, но и рискуем существенно исказить оставшуюся, поскольку исчезнувшие клетки могли бы дать новые организмы в видимой части вселенной. Во *втором* случае клетки, пересекающие границы поля, будут появляться с *противоположной*

стороны и деформировать популяцию самым неожиданным образом. Как говорится, хрен редьки не слаще, но зато у игрока есть выбор…

Максимальные размеры поля в этой версии такие:

```
const
  NAME_PROG = 'ЖИЗНЬ';
  //размеры поля в клетках:
  POLE_WIDTH = 75; // ширина
  POLE_HEIGHT = 54;// высота
```


Вы, конечно, можете сделать поле и б**о**льших размеров, но тогда его придётся прокручивать, чтобы обозреть всё происходящее в мире.

Главное место на форме займёт *сетка-поле*:

```
Name = dgPole
Color = clBlack
ColCount = 75
RowCount = 54
DefaultColWidth = 8
DefaultRowHeight = 8
GridLineWidth = 0
Hint = |Левая кнопка - поставить, правая - убрать
```

Прокомментировать уместно только *цвет* вселенной. Я выбрал *чёрный* не только потому, что и наша Вселенная того же цвета, но – главным образом – потому, что на чёрном фоне другие цвета выглядят более контрастными.

В целом *интерфейс* программы может быть таким (Рис. 6.1).

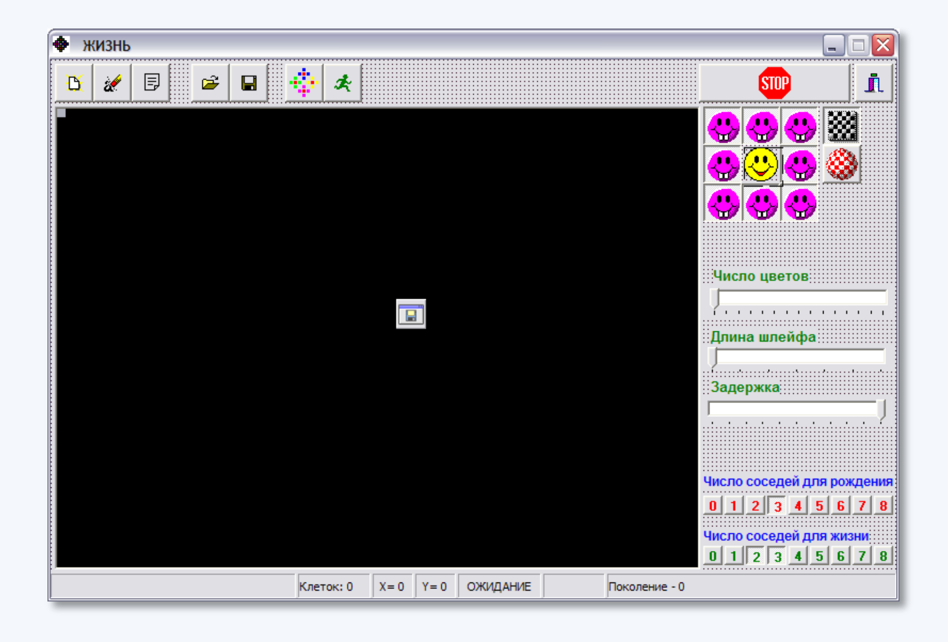


Рис. 6.1. Вселенная в начале *Жизни*

А это наше приложение в действии (Рис. 6.2)

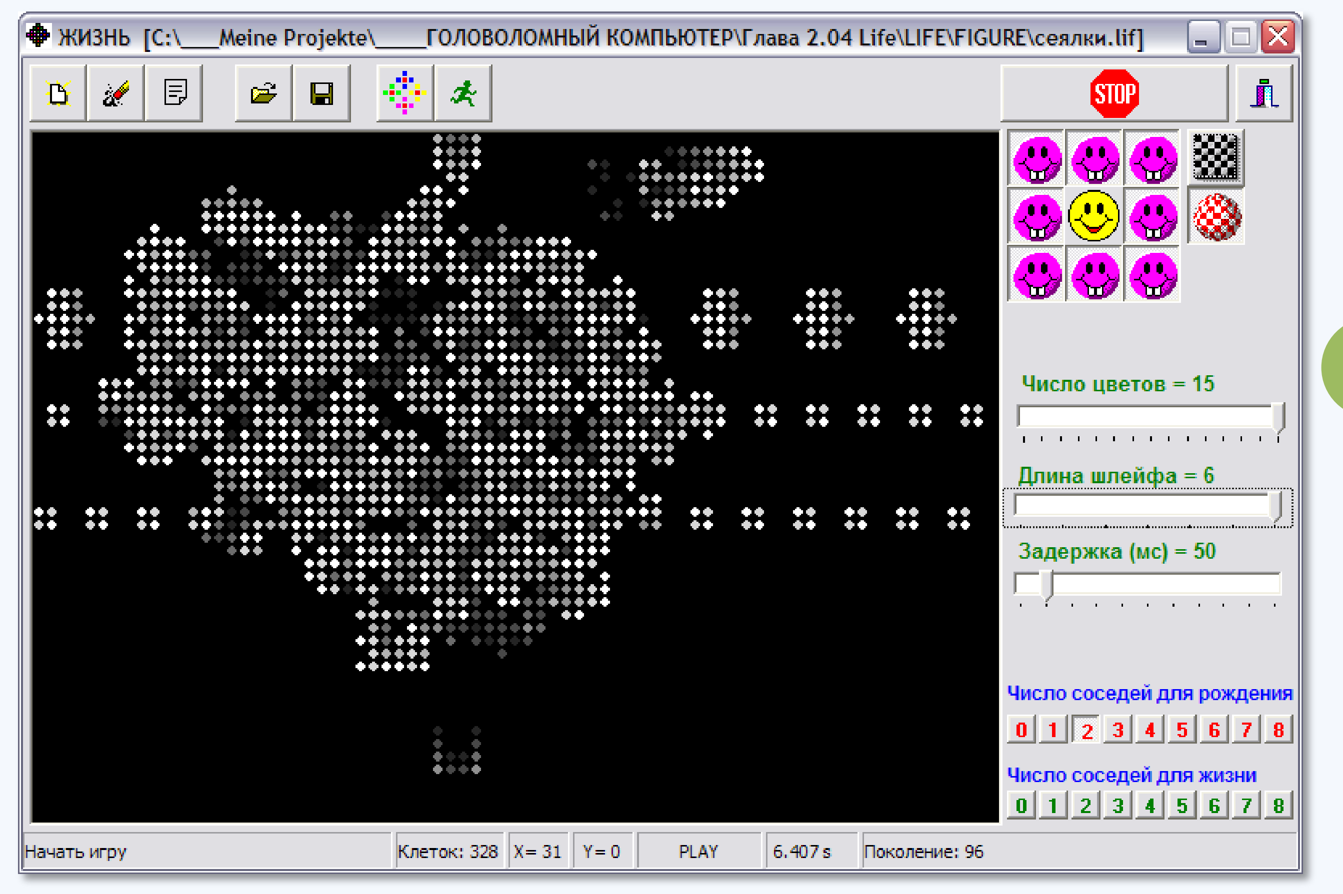


Рис. 6.2. Время пошло!

Наш мир возникает вместе с созданием формы приложения:

```
///////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////
//                         ФОРМА                             //
///////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////
//СОЗДАТЬ ФОРМУ
procedure TForm1.FormCreate(Sender: TObject);
var i: integer;
begin
  //очистить поле:
  ClearPole;
  //загрузить курсоры:
  Screen.Cursors[crKistj]:= LoadCursor(hinstance,'kistj');
  Screen.Cursors[crHand]:= LoadCursor(hinstance,'hand');
  dgPole.Cursor :=crHand;

  RandSeed:=6; //4
  //задать цвета клеток опр. возраста:
  AgeColor[0]:=COLOR_PUSTO;
  AgeColor[1]:=COLOR_KLETKA;
```

```
  for i:=2 to MAX_COLOR do AgeColor[i]:= RGB(Random(255),
Random(255), Random(255));
  //задать цвета шлейфа:
  tbShleifChange(Self);
  //задать скорость игры:
  tbDelayChange(Self);
  //максимальная продолжительность жизни клеток:
  MAX_AGE:=MAX_COLOR;
  tbColor.max:=MAX_COLOR;
  //задать число цветов клеток:
  tbColorChange(Self);
  //русская раскладка клавиатуры:
  LoadKeyboardLayout('00000419',KLF_ACTIVATE);
end;
```


Сначала он будет абсолютно чист:

```
//ОЧИСТИТЬ ПОЛЕ
procedure TForm1.ClearPole;
var
  i,j: integer;
  hc, wc:  integer;
begin
  if GameStatus='PLAY' then exit;
  nGen:=0;
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
  hc:= dgPole.DefaultRowHeight;
  wc:= dgPole.DefaultColWidth;
  dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
  For i:= 0 To POLE_WIDTH-1 do
    for j:= 0 to  POLE_HEIGHT - 1 do begin
      dgPole.Canvas.Ellipse (i*wc,j*hc,i*wc+wc,j*hc+hc);
      masPole[i,j]:= NUM_PUSTO;
   end;
   vsegoKl_inFig:=0;
   statusbar1.Panels[1].text:='Клеток: 0';
   cellmouse.x:=-1;
   cellmouse.y:=-1;
end;
```

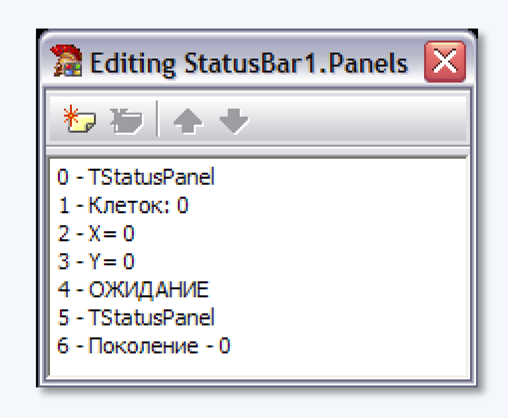
В переменной *nGen* мы будем хранить номер текущего поколения. Для начальной колонии это значение равно *нулю*:

```
//номер поколения:
  nGen: integer=0;
```

Вся информация о текущем состоянии популяции отображается в компоненте *StatusBar1* TStatusBar:

```
AutoHint=True
Height=25
```

Panels:




```
                          Width=230
Text=Клеток: 0            Width=70
Text=X= 0                 Width=40
Text=Y= 0                 Width=40
Text=ОЖИДАНИЕ             Width=80
                          Width=60
Text=Поколение - 0        Width=50
```



```
ShowHint=True
```

Текстовую информацию очень удобно выводить в нужную *панель* этого компонента. Так как все панели образуют *массив*, то достаточно указать только индекс панели в этом массиве. Например, так указывается номер текущего поколения:

```
statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
```

В *нулевой* панели выводятся *подсказки*, которые также появляются возле компонента, когда на нём останавливается курсор мыши. Если вы хотите, чтобы эти подсказки были разными, то напишите их с разделительной чертой: *Очистить| Очистить поле*. Подсказка *до* черты будет появляться у компонента, *после* черты – в строке состояния.

В *первой* панели указывается число клеток в текущей популяции. Во *второй* и *третьей* – координаты мышки на поле, что облегчает рисование новых колоний. В *четвёртой* – состояние игры. В *пятой* – время формирования нового поколения. И в последней, *шестой*, - номер текущего поколения.

После вывода информации мы очищаем игровое поле. Для этого в каждой клетке поля кистью цвета

```
dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
```

мы рисуем маленькие *кружки*:

```
dgPole.Canvas.Ellipse (i*wc,j*hc,i*wc+wc,j*hc+hc);
```



Было бы проще *полностью* закрашивать клетки, но в этом случае отдельные «особи» сливаются в одну область и плохо различимы (конечно, их можно отделить друг от друга вертикальными и горизонтальными линиями сетки, но всё равно получится хуже…).

Параллельно мы заполняем массив константой *NUM_PUSTO*, которая соответствует *пустой* клетке поля:

```
masPole[i,j]:= NUM_PUSTO;
```

Объявим *константы*:

```
const
  //цвет пустой клетки на поле:
  COLOR_PUSTO: TColor= clBlack;
  //цвет новой клетки фигуры на поле:
  COLOR_KLETKA: TColor= clWhite;
  //пустая клетка в masPole:
  NUM_PUSTO = 0;
  //чёрная клетка в masPole:
  NUM_KLETKA = 1;
```


Таким образом, клетки без организмов будут *чёрными* (они просто сливаются с чёрным фоном), с организмами – *белые*.

Массив *поля*

```
type TPole = array[-1..POLE_WIDTH,-1..POLE_HEIGHT]of Integer;
//(добавляем по одной клетке по периметру поля)

var
//массивы, в кот. хранится копия поля:
  masPole, masPole2: TPole;
```

мы задаём так, чтобы за его пределами (по периметру) была ещё одна клетка. В игре эти дополнительные клетки не участвуют, но облегчают формирование нового поколения, так как избавляют нас от лишних проверок.

Последние две строки процедуры *TForm1.ClearPole*

```
   cellmouse.x:=-1;
   cellmouse.y:=-1;
```

говорят о том, что курсор мыши находится за пределами поля. Эти координаты используются при рисовании исходной популяции:

```
Var
  //координаты клетки с курсором:
  cellMouse: TPoint;
```

Возвращаемся к процедуре *TForm1.FormCreate*. Чтобы при рисовании курсор больше соответствовал своему назначению, будем изменять его *форму*:

```
  //загрузить курсоры:
  Screen.Cursors[crKistj]:= LoadCursor(hinstance,'kistj');
  Screen.Cursors[crHand]:= LoadCursor(hinstance,'hand');
  dgPole.Cursor:=crHand;
```

Форма курсора на игровом поле в *обычном* режиме (Рис. 6.3).

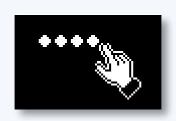


Рис. 6.3. Обычный курсор

В режиме *рисования* клеток на игровом поле курсор превращается в кисточку (Рис. 6.4).

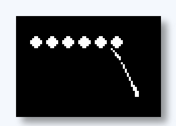


Рис. 6.4. Рисовальный курсор

Нужно также позаботиться о *константах*:

```
const
  //курсоры:
  crHand :integer= 4;
  crStrelka :integer= 5;
  crUkaska: integer= 6;
  crKistj :integer= 7;
```

Все курсоры хранятся в файле ресурсов, который следует подключить к проекту:

```
uses MemoUnit, LoadUnit;

{$R *.DFM}
{$R MyCursors.res}
```

В этой программе используются не все имеющиеся курсоры, так что вы можете выбирать любые!

В классической игре клетки, как и мёд у Винни-Пуха, либо есть, либо нет, но если вам захочется разнообразия, то вы можете раскрашивать клетки в разные цвета:

```
  //максимальное число цветов клеток:
  MAX_COLOR=15;
  MAX_SHLEIF= 6;
```

Первая константа задаёт максимальное число цветов живых клеток, *вторая* – мёртвых. В переменной *MAX_AGE*

```
  //максимальная продолжительность жизни клеток:
  MAX_AGE: integer;
```


хранится число цветов живых клеток. Вы только должны иметь в виду, что клетки могут жить вечно, поэтому никаких цветов не хватит, чтобы их раскрасить в разные цвета, и в программе по достижении клеткой возраста *MAX_AGE* она вновь становится «молодой» и окрашивается в цвет *NUM_KLETKA.* И так далее, по кругу.

```
  RandSeed:=6; //4
  //задать цвета клеток опр. возраста:
  AgeColor[0]:=COLOR_PUSTO;
  AgeColor[1]:=COLOR_KLETKA;
  for i:=2 to MAX_COLOR do AgeColor[i]:= RGB(Random(255),
Random(255), Random(255));
  //задать цвета шлейфа:
  tbShleifChange(Self);
  //задать скорость игры:
  tbDelayChange(Self);
  //максимальная продолжительность жизни клеток:
  MAX_AGE:=MAX_COLOR;
  tbColor.max:=MAX_COLOR;
  //задать число цветов клеток:
  tbColorChange(Self);
```

До игры и по ее ходу вы можете изменять количество цветов живых клеток *движком tbColor* TTrackBar:

```
Hint=Количество цветов|Максимальный возраст особи
Max=15
Min=1
```

```
/////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////
//                         ДВИЖКИ                          //
/////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////

//ИЗМЕНИТЬ КОЛИЧЕСТВО ЦВЕТОВ КЛЕТОК
procedure TForm1.tbColorChange(Sender: TObject);
begin
  MAX_AGE:=tbColor.Position;
```

```
  lblColor.Caption:= 'Число цветов = '+inttostr(MAX_AGE);
end;
```

Значение переменной *MAX_AGE* выводится в *метке lblColor*.

Аналогично можно изменять и количество цветов умерших клеток (длину шлейфа) *движком tbShleif* TTrackBar:

Hint= Шлейф|Ушедшие поколения
Max=6
Min=0


```
//ЗАДАТЬ ЧИСЛО ЦВЕТОВ В ШЛЕЙФЕ
procedure TForm1.tbShleifChange(Sender: TObject);
var i,j: integer;
begin
  shleif:=tbShleif.Position;
  if shleif>0 then
    for i:= 1 to shleif do begin
      j:=255 div (Shleif+1) * i;
      AgeColor[-(shleif-i+1)]:= RGB(j,j,j)
    end;
  lblShleif.Caption:= 'Длина шлейфа = '+inttostr(shleif);
end;
```

И последний *движок* - *tbDelay* - управляет скоростью смены поколений на экране:

Hint= Задержка|Пауза между поколениями
Max=10
Min=0

```
//ЗАДАТЬ ЗАДЕРЖКУ ПРИ ВЫВОДЕ НОВОГО ПОКОЛЕНИЯ
procedure TForm1.tbDelayChange(Sender: TObject);
begin
  Delay:=tbDelay.Position * 50;
  lblDelay.Caption:= 'Задержка (мс) = '+inttostr(Delay)
end;
```

После создания поля на нём нет ни одной живой клетки. Такой мир развиваться не может, и мы должны произвести начальный посев. Сделать это можно двумя способами: *нарисовать* новую популяцию прямо на поле или *загрузить* готовую с диска (несколько заготовок вы найдёте в *папке Figure*).

Начнём с рисования. Нажмём кнопку мышки на поле:

```
//НАЖАТЬ КНОПКУ МЫШКИ НА ПОЛЕ
```

```
procedure TForm1.dgPoleMouseDown(Sender: TObject; Button:
TMouseButton; Shift: TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  hc, wc:  integer;
begin
  if GameStatus='PLAY' then exit;
  //изменить форму курсора:
  screen.Cursor:=TCursor(crKistj);
  dgPole.Cursor :=TCursor(crKistj);
  screen.Cursor:=TCursor(crDefault);

  dgPole.MouseToCell(x,y,ACol,ARow);
  //размеры клеток поля:
  hc:= dgPole.DefaultRowHeight;
  wc:= dgPole.DefaultColWidth;
  //если нажата левая кнопка - белая клетка:
  if ssLeft in shift then
  begin
    if masPole[ACol, ARow]= NUM_PUSTO
    then inc(vsegoKl_inFig);
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:=NUM_KLETKA;
    //закрасить клетку:
    dgPole.Canvas.Brush.Color:=COLOR_KLETKA;
    dgPole.Canvas.Ellipse (ACol*wc,ARow*hc,ACol*wc+wc,ARow*hc+hc);
  end;
  //если нажата правая кнопка - чёрная клетка:
  if ssRight in shift then
  begin
    if masPole[ACol, ARow]= NUM_KLETKA
    then dec(vsegoKl_inFig);
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:= NUM_PUSTO;
    //закрасить клетку:
    dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
    dgPole.Canvas.Ellipse (ACol*wc,ARow*hc,ACol*wc+wc,ARow*hc+hc);
  end;
  //теперь клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
end;
```


Курсор мышки превратится в кисточку, напоминая о наших намерениях. В клетку *(ACol,ARow)* мы помещаем новый организм - если нажата *левая* кнопка мышки:

```
masPole[ACol, ARow]:=NUM_KLETKA;
```

или убираем организм из клетки, если нажата *правая* кнопка:

```
masPole[ACol, ARow]:= NUM_PUSTO;
```

Одновременно мы рисуем кружки белого или чёрного цвета и подсчитываем число клеток в популяции.

Так можно «высаживать» организмы «поштучно», но иногда требуется засеять большие площади. Для этого, не отпуская кнопку мышки, перемещаем её по полю:

```
//ПЕРЕМЕЩАТЬ МЫШКУ ПО ПОЛЮ - РИСОВАТЬ ФИГУРУ
procedure TForm1.dgPoleMouseMove(Sender: TObject; Shift:
TShiftState; X, Y: Integer);
var
  ACol,ARow: integer;
  hc, wc: integer;
begin
  if GameStatus='PLAY' then exit;
  //размеры клеток поля:
  hc:= dgPole.DefaultRowHeight;
  wc:= dgPole.DefaultColWidth;
  dgPole.MouseToCell(x,y,ACol,ARow);
  statusbar1.Panels[2].text:='X= '+inttostr(ACol);
  statusbar1.Panels[3].text:='Y= '+inttostr(ARow);
  //если нажата левая кнопка - белая клетка:
  if (ssLeft in shift) and ((cellmouse.x<>ACol) or
(cellmouse.y<>ARow))then
  begin
    if masPole[ACol, ARow]= NUM_PUSTO then inc(vsegoKl_inFig);
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:=NUM_KLETKA;
    //закрасить клетку:
    dgPole.Canvas.Brush.Color:=COLOR_KLETKA;
    dgPole.Canvas.Ellipse (ACol*wc,ARow*hc,ACol*wc+wc,ARow*hc+hc);
  end;
  //если нажата правая кнопка - чёрная клетка:
  if (ssRight in shift) and ((cellmouse.x<>ACol) or
(cellmouse.y<>ARow))then
  begin
    if masPole[ACol, ARow]= NUM_KLETKA then dec(vsegoKl_inFig);
    //занести в массив цвет клетки:
    masPole[ACol, ARow]:= NUM_PUSTO;
    //закрасить клетку:
    dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
    dgPole.Canvas.Ellipse (ACol*wc, ARow*hc, ACol*wc+wc,
ARow*hc+hc);
  end;
  //теперь клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
end;
```

Процедура рисования живности практически та же самая, только дополнительно мы показываем в строке состояния координаты мышки на поле.

Осталось отпустить кнопку мышки и закончить рисование:

```
//ОТПУСТИТЬ КНОПКУ МЫШКИ
procedure TForm1.dgPoleMouseUp(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  //изменить форму курсора:
  screen.Cursor:=TCursor(crHand);
  dgPole.Cursor :=TCursor(crHand);
  screen.Cursor:=TCursor(crDefault);
end;
```


Здесь всё понятно «без слов».

Если вам потребуется убрать всю популяцию разом, воспользуйтесь *кнопкой sbtClearPole* (**Hint**= Очистить| Очистить поле):

```
//КНОПКА Очистить поле
procedure TForm1.sbtClearPoleClick(Sender: TObject);
begin
  ClearPole;
end;
```

Аналогично, только более радикально действует *кнопка sbtNewFig* (**Hint**=Новая фигура| Новая фигура):

```
//КНОПКА Новая фигура
procedure TForm1.sbtNewFigClick(Sender: TObject);
begin
  if GameStatus='PLAY' then exit;
  //название новой фигуры:
  NameFig:= 'temp';
  caption:= NAME_PROG + '  [' + NameFig + ']';
  //очистить поле:
  ClearPole;
  //стереть комментарии:
  frmMemo.memo1.Clear;
  frmMemo.memo1.SetFocus;
  //номер поколения:
  nGen:=0;
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
end;
```

Конечно, вам не терпится посмотреть, как же будет эволюционировать нарисованная вами популяция организмов. Что ж, это вполне законное желание. Сейчас мы и займёмся проблемами эволюции, но прежде мы должны позаботиться о настройке параметров окружающей среды, дабы получить именно то, что мы и предвкушаем увидеть.

По умолчанию мир развивается по *классическим* законам, то есть для рождения новой клетки необходимо иметь *трёх* и только трёх соседей, а для продолжения жизни – *двух* или *трёх*. Эти данные уместно хранить в двух массивах:


```
var
  //массив числа соседей,
  //необходимых для рождения клетки:
  Birth: array[0..8] of boolean;
  //массив числа соседей,
  //необходимых для продолжения жизни:
  Life: array[0..8] of boolean;
```

По правилам *Жизни*, *Birth[3]=True*, *Birth[0..2, 4..8]=False*; *Life[2,3]=True*, *Life[0..1, 4..8]=False*. Чтобы игрок мог изменять правила игры, установим на форме по 9 кнопок (Рис. 6.5).



Рис. 6.5. «Правильные» кнопки

Первый ряд кнопок служит для назначения числа соседей, необходимых для зарождения новых клеток, *второй* – для продолжения жизни уже имеющихся клеток. Нажатые клетки показывают условия эволюции:

```
sbtBirth0.. sbtBirth8
Hint=Рождение|Рождение новой клетки
AllowAllUp=True
GroupIndex=10..18
Down=False (sbtBirth3 Down=True)
Tag=0..8
```

```
//КНОПКИ Рождение новой клетки
//Число соседей, необходимых для рождения новой клетки
procedure TForm1.sbtBirth0Click(Sender: TObject);
begin
  if (sender as Tspeedbutton).down
  then Birth[(sender as Tspeedbutton).tag]:=true
```

```
  else Birth[(sender as Tspeedbutton).tag]:=false
end;
```

sbtLife0.. sbtLife8
Hint=Сохранение|Клетка остаётся живой
AllowAllUp=True
GroupIndex=20..28
Down=False (**sbtLife2 sbtLife3 Down**=True)
Tag=0..8


```
//КНОПКИ Сохранение - Клетка остаётся живой
//Число соседей, необходимых для продолжения жизни
procedure TForm1.sbtLife0Click(Sender: TObject);
begin
  if (sender as Tspeedbutton).down
  then Life[(sender as Tspeedbutton).tag]:=true
  else Life[(sender as Tspeedbutton).tag]:=false
end;
```

Обычно все восемь соседних клеток учитываются при подсчёте живых соседей, однако вы можете самостоятельно назначить соседей для клеток. Установим на форме еще 8 кнопок:

Name=sbtSosedLU **AllowAllUp**=True **Down**=True **GroupIndex**=1 **Hint=**Сосед
Name=sbtSosedU **AllowAllUp**=True **Down**=True **GroupIndex**=2 **Hint=**Сосед
Name=sbtSosedRU **AllowAllUp**=True **Down**=True **GroupIndex**=3 **Hint=**Сосед
Name=sbtSosedL **AllowAllUp**=True **Down**=True **GroupIndex**=4 **Hint=**Сосед
Name=sbtSosedR **AllowAllUp**=True **Down**=True **GroupIndex**=5 **Hint=**Сосед
Name=sbtSosedLD **AllowAllUp**=True **Down**=True **GroupIndex**=6 **Hint=**Сосед
Name=sbtSosedD **AllowAllUp**=True **Down**=True **GroupIndex**=7 **Hint=**Сосед
Name=sbtSosedRD **AllowAllUp**=True **Down**=True **GroupIndex**=8 **Hint=**Сосед

Как видите, первоначально все кнопки *нажаты*, что полностью соответствует классическим правилам игры (Рис. 6.6).

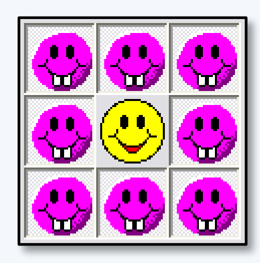


Рис. 6.6. «Соседские» кнопки

Теперь давайте вспомним об ограниченности мира. В начале главы мы уже говорили об этом и придумали 2 способа конструирования вселенной в игре. Для выбора нужной геометрии пространства нам потребуются две *кнопки-переключателя* (Рис. 6.7).

```
Name=sbtFlat Down=True GroupIndex=9 Hint=Плоский мир
Name=sbtSphere Down=False GroupIndex=9 Hint=Сферический мир
```

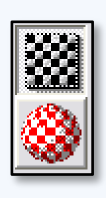
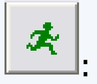


Рис. 6.7. Искажаем пространство!

Популяция развивается хоть и дискретно, но довольно быстро, так что бывает трудно уследить за сменой поколений. Конечно, движком *tbDelay* можно изменить скорость развития жизни, но всё равно за одним поколением сразу же следует другое. Поэтому мы установим *кнопку sbtStep* :

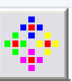
```
AllowAllUp=True
Down=False
GroupIndex=30
Hint=Один шаг|Пошаговая игра
```

Когда эта кнопка *нажата*

```
//КНОПКА Пошаговая игра
procedure TForm1.sbtStepClick(Sender: TObject);
begin
  step:= sbtStep.down
end;
```

переменная *step* равна *TRUE*, и на экран выводится только *одно* следующее поколение.

```
var
//режим вывода поколений:
  //TRUE - пошаговый
  //FALSE - непрерывный
  step: boolean= false;
```

А затем нужно снова нажать кнопку *sbtGo*  (**Hint**=Игра|Начать игру).

```
//КНОПКА Начать игру
procedure TForm1.sbtGoClick(Sender: TObject);
var
  i,j: integer;
  index: integer;
  h, w:  integer;
```

```
time,time0: single;
s: string;
hc, wc: integer;

//сформировать следующее поколение
procedure NewGen;
  begin
    //подсчитать число соседей:
    index:=0;
    //левый верхний сосед:
    if sbtSosedLU.Down and (masPole[i-1,j-1]>0)then inc(index);
    //верхний сосед:
    if sbtSosedU.Down and (masPole[i,j-1]>0)then inc(index);
    //правый верхний сосед:
    if sbtSosedRU.Down and (masPole[i+1,j-1]>0)then inc(index);
    //левый сосед:
    if sbtSosedL.Down and (masPole[i-1,j]>0)then inc(index);
    //правый сосед:
    if sbtSosedR.Down and (masPole[i+1,j]>0)then inc(index);
    //левый нижний сосед:
    if sbtSosedLD.Down and (masPole[i-1,j+1]>0)then inc(index);
    //нижний сосед:
    if sbtSosedD.Down and (masPole[i,j+1]>0) then inc(index);
    //правый нижний сосед:
    if sbtSosedRD.Down and (masPole[i+1,j+1]>0)then inc(index);
  end;

//скроллинг поля
procedure scroll;
var i_s,j_s: integer;
  begin
    //копируем TopRow --> PostLastRow:
    for i_s:=0 to POLE_WIDTH-1 do
      masPole[i_s,POLE_HEIGHT]:=masPole[i_s,0];
    //копируем LastRow --> PreTopRow:
    for i_s:=0 to POLE_WIDTH-1 do //
      masPole[i_s,-1]:=masPole[i_s,POLE_HEIGHT-1];
    //копируем LeftCol --> PostRightCol:
    for j_s:=0 to POLE_HEIGHT-1 do //
      masPole[POLE_WIDTH,j_s]:=masPole[0,j_s];
    //копируем Right --> PreLeftCol:
    for j_s:=0 to POLE_HEIGHT-1 do //
      masPole[-1,j_s]:=masPole[POLE_WIDTH-1,j_s];
    //копируем левую верхнюю угловую клетку:
    masPole[POLE_WIDTH,POLE_HEIGHT]:=masPole[0,0];
    //копируем правую верхнюю угловую клетку:
    masPole[-1,POLE_HEIGHT]:=masPole[POLE_WIDTH-1,0];
    //копируем левую нижнюю угловую клетку:
    masPole[POLE_WIDTH,-1]:=masPole[0,POLE_HEIGHT-1];
    //копируем правую нижнюю угловую клетку:
    masPole[0,0]:=masPole[POLE_WIDTH-1,POLE_HEIGHT-1];
  end;
```

```
begin
  if GameStatus= 'PLAY' then exit;
  //игра началась:
  Outstatus('PLAY');

  //число соседей, позволяющих продолжить жизнь:
  for i:=0 to 8 do
    Life[i]:=(FindComponent('sbtLife'+inttostr(i)) as
TSpeedButton).down;
  //число соседей, необходимых для рождения:
  for i:=0 to 8 do
    Birth[i]:=(FindComponent('sbtBirth'+inttostr(i)) as
TSpeedButton).down;
  //засечь начало игры:
  time0:=GetTickCount();
  //размеры поля:
  h:= POLE_HEIGHT - 1;
  w:= POLE_WIDTH-1;
  hc:= dgPole.DefaultRowHeight;
  wc:= dgPole.DefaultColWidth;

//следующее поколение:
repeat
  inc(nGen);
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
  vsegoKl_inFig:=0;

  //если поле замкнутое, копируем крайние строки и столбцы:
  if sbtSphere.Down then scroll;

  //просматриваем поле и заполняем вспомогательный массив:
  For j := 0 To h do //- по ширине
    for i := 0 to w do begin //- по высоте
      NewGen;
      if masPole[i,j]>0 then //- живая клетка -> стареет или
умирает
      begin

      if Life[index] then //- продолжает жить
        begin
          inc(vsegoKl_inFig); //- ещё одна клетка на поле
          if masPole[i,j]>=MAX_AGE then
             masPole2[i,j]:= NUM_KLETKA //- постаревшая клетка
          else masPole2[i,j]:=masPole[i,j]+1;
        end
        else //- стареет или умирает -> пустая клетка
        if Shleif>0 then masPole2[i,j]:= -1
        else masPole2[i,j]:= NUM_PUSTO;
        //закрасить клетку соотв. цветом:
        dgPole.Canvas.Brush.Color:= AgeColor[masPole2[i,j]];
        dgPole.Canvas.Ellipse (i*wc,j*hc,(i+1)*wc,(j+1)*hc);
      end
```

```
      else //пустая клетка поля(или клетка шлейфа)
      begin
        if Birth[index] then //- рождается
        begin
          masPole2[i,j]:= NUM_KLETKA; //- новорождённая клетка
          inc(vsegoKl_inFig);
          //закрасить клетку соотв. цветом:
          dgPole.Canvas.Brush.Color:= AgeColor[masPole2[i,j]];
          dgPole.Canvas.Ellipse (i*wc,j*hc,i*wc+wc,j*hc+hc);
        end
        else begin //- стареет или умирает -> пустая клетка
          masPole2[i,j]:= NUM_PUSTO;
          if Shleif>0 then
            if masPole[i,j]<0 then begin
               masPole2[i,j]:= masPole[i,j]-1;
               if masPole2[i,j]<-SHLEIF
               then masPole2[i,j]:= NUM_PUSTO
               else;
               //закрасить клетку соотв. цветом:
               dgPole.Canvas.Brush.Color:=
AgeColor[masPole2[i,j]];
               dgPole.Canvas.Ellipse
(i*wc,j*hc,(i+1)*wc,(j+1)*hc);
            end
            else masPole2[i,j]:= NUM_PUSTO //- пустая клетка ->
цвет не меняет
          else begin  //Shleif=0
            masPole2[i,j]:= NUM_PUSTO;
            if masPole[i,j]<0 then begin  //- стереть следы
              dgPole.Canvas.Brush.Color:= COLOR_PUSTO;
              dgPole.Canvas.Ellipse (i*wc,j*hc,(i+1)*wc,(j+1)*hc)
            end;
          end;
        end;
      end;
  end;
  //копируем вспомогательный массив в основной:
  masPole:= masPole2;

  sleep(Delay);
  //теперь клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
  //время игры:
  time:=(GetTickCount()-time0)/1000;
  str(time:6:3,s);
  statusbar1.Panels.Items[5].text:=s + ' s';

  //если была нажата кнопка STOP, то закончить
  //формирование новых поколений:
  application.ProcessMessages;
  if flgExit=true then begin
    flgExit:=false;
    Outstatus('ОЖИДАНИЕ');
```

```
    exit
  end;
until (vsegoKl_inFig=0) or step;
   Outstatus('ОЖИДАНИЕ');
end;
```

Эта процедура, безусловно, самая важная в программе – именно она производит все «манипуляции» с клетками текущей популяции.

Сначала нужно определить, по каким законам будет развиваться жизнь:


```
  //число соседей, позволяющих продолжить жизнь:
  for i:=0 to 8 do
    Life[i]:=(FindComponent('sbtLife'+inttostr(i)) as
TSpeedButton).down;
  //число соседей, необходимых для рождения:
  for i:=0 to 8 do
    Birth[i]:=(FindComponent('sbtBirth'+inttostr(i)) as
TSpeedButton).down;

. . .

  //если поле замкнутое, копируем крайние строки и столбцы:
  if sbtSphere.Down then scroll;
```

Затем в цикле

```
//следующее поколение:
repeat

. . .

until (vsegoKl_inFig=0) or step;

Outstatus('ОЖИДАНИЕ');
```

генерируются новые поколения клеток - до тех пор, пока колония не вымрет. Другая возможность окончания цикла – активный пошаговый режим; в этом случае процедура завершается и переходит в режим ввода команд уже после формирования первого поколения. Вы также можете «насильственно» закончить работу процедуры, если нажмёте *кнопку sbtStop* STOP (**Hint** =Остановить|Остановить игру):

```
//КНОПКА Остановить игру
procedure TForm1.sbtStopClick(Sender: TObject);
begin
```

```
  if GameStatus<>'PLAY' then exit;
  flgExit:=true
end;
```

В этой процедуре-обработчике переменной *flgExit* присваивается значение *TRUE*:

```
 var
 //=TRUE - остановить игру:
  flgExit: boolean= False;
```


При формировании нового поколения в процедуре *TForm1.sbtGoClick* проверяется эта переменная, и как только она становится равной *TRUE*, процедура завершается:

```
  application.ProcessMessages;
  if flgExit=true then begin
    flgExit:=false;
    Outstatus('ОЖИДАНИЕ');
    exit
  end;
```

Обратите также внимание на переменную *GameStatus:*

```
var
  GameStatus: string='';
```

Она отражает *текущее* состояние программы, которое анализируется при нажатии кнопок. Если программа находится в состоянии ожидания ввода команд, то они выполняются. Если работает игровая процедура, то все попытки вмешательства в эволюцию игнорируются.

Для каждой клетки поля в процедуре *NewGen*

```
  //сформировать следующее поколение
  procedure NewGen;

  . . .
```

мы подсчитываем в переменной *index* количество соседей и поступаем в соответствии с законами жизни. Если *Life[index]=TRUE* и в клетке находится живая особь (*masPole[i,j]>0*), то она продолжает жить, но стареет (клетки от старости не умирают, а только окрашиваются в другой цвет, чтобы было удобнее наблюдать за развитием популяции). В противном случае особь умирает, оставляя после себя пустую, *чёрную* клетку (если игрок задал положительную длину шлейфа, то клетка будет «чернеть»

постепенно, хотя умрёт мгновенно; причина здесь та же, что и при старении клеток):

```
  NUM_PUSTO;
  //закрасить клетку соотв. цветом:
  dgPole.Canvas.Brush.Color:= AgeColor[masPole2[i,j]];
  dgPole.Canvas.Ellipse (i*wc,j*hc,(i+1)*wc,(j+1)*hc);
```

Если *Birth[index] =TRUE* и клетка *пустая*, то в ней возникает новая жизнь:

```
  //закрасить клетку соотв. цветом:
  dgPole.Canvas.Brush.Color:= AgeColor[masPole2[i,j]];
  dgPole.Canvas.Ellipse (i*wc,j*hc,i*wc+wc,j*hc+hc);
```


Здесь важно учесть уже умершие клетки, которые остаются на экране в качестве шлейфа. С каждым новым поколением они уходят всё дальше и дальше в небытиё и по достижении установленной длины шлейфа исчезают навсегда, что по-своему очень печально…

И последнее замечание. Поскольку клетки нового поколения рождаются и умирают одновременно, то мы ничего не можем изменить в массиве *masPole[]* с текущим поколением, пока не просчитаем следующее. Поэтому новую популяцию следует формировать в *дополнительном* массиве *masPole2[],* а затем целиком скопировать в основной:

```
  //копируем вспомогательный массив в основной:
  masPole:= masPole2;
```

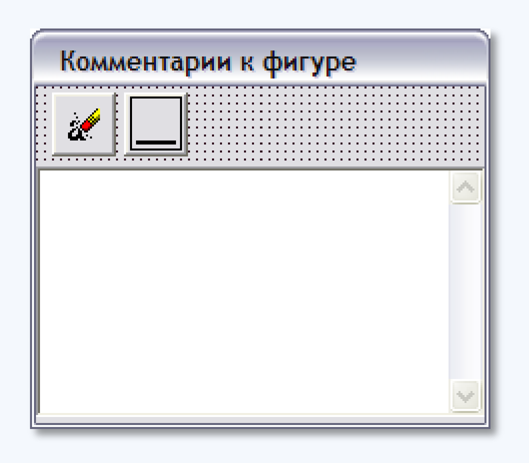
Как видите, всё достаточно просто, а было бы ещё проще, если бы не раскрашивание стареющих и умерших клеток в разные цвета…

Игровой статус отображается внизу, в строке состояния:

```
/////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////
//                    СТРОКА СОСТОЯНИЯ                     //
/////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////

procedure TForm1.OutStatus(s:string);
begin
  GameStatus:=s;
  statusbar1.Panels[4].text:= GameStatus
end;
```

Свои соображения по поводу новой фигуры (популяции) вы можете записать в форме *frmMemo* (Рис. 6.8).



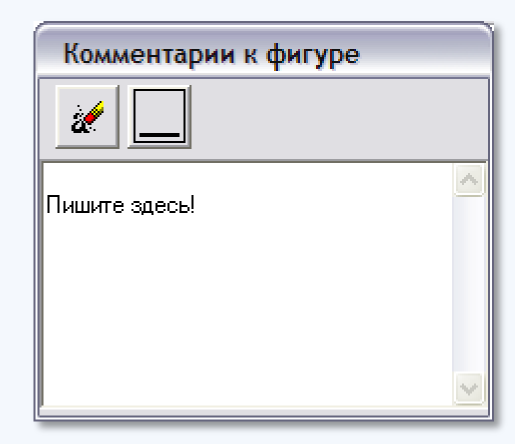


Рис. 6.8. «Замечательная» форма

Она появляется справа от главного окна приложения при старте программы. Предполагается, что ваши комментарии будут на русском языке:

```
  //русская раскладка клавиатуры:
  LoadKeyboardLayout('00000419',KLF_ACTIVATE);
```

Эту форму можно свернуть, нажав на ней *кнопку sbtMinimize* (**Hint**=Свернуть|Свернуть окно), а при необходимости вновь возвратить на экран *кнопкой sbtMemo* (**Hint**=Комментарий|Развернуть комментарий к фигуре), которая находится на главной форме приложения (можно и просто дважды щёлкнуть по заголовку свёрнутой формы):

```
//КНОПКА Развернуть комментарий к фигуре
procedure TForm1.sbtMemoClick(Sender: TObject);
begin
  frmMemo.WindowState:=wsNormal
end;
```

Ваши записи хранятся в *компоненте Memo1* TMemo (**Hint**=Комментарий| Комментарии к фигуре) и будут записаны в файл вместе с фигурой.

Кроме возможностей по редактированию текста, предоставляемых самим компонентом, можно добавить ещё и полное стирание текста, которое производится при нажатии на *кнопку sbtClear* (**Hint**=Очистить| Очистить):

```
implementation

uses lifeunit;
```

```
{$R *.DFM}

procedure TfrmMemo.sbtClearClick(Sender: TObject);
begin
  frmMemo.memo1.Clear;
  frmMemo.memo1.SetFocus
end;

procedure TfrmMemo.sbtMinimizeClick(Sender: TObject);
begin
  frmMemo.WindowState:=wsMinimized
end;
```


Чтобы записать всю информацию о популяции, нужно нажать *кнопку sbtSaveFig* (**Hint**= Записать|Записать фигуру). Как обычно в таких случаях, мы добавляем для этого на форму *компонент SaveDialog1* TSaveDialog.

```
//КНОПКА Записать фигуру
procedure TForm1.sbtSaveFigClick(Sender: TObject);
var
  F: textfile;
  fn,s: string;
  i,j: integer;
begin
  if GameStatus='PLAY' then exit;
  savedialog1.DefaultExt:='txt';
  savedialog1.Filter:='Фигуры (*.' + ext + ')|*.' + ext;
  savedialog1.FilterIndex:=1;
  s:=extractfilepath(application.exename)+'FIGURE\';
  savedialog1.InitialDir:= s;
  savedialog1.Title:='Запишите фигуру на диск';
  savedialog1.filename:= NameFig;
  if not savedialog1.Execute then exit;
  fn:= savedialog1.filename; {имя конечного файла}
  NameFig:=fn;
  assignfile(f,fn);
  rewrite(f);
  //записать фигуру:
  writeln (f,vsegoKl_inFig);     //- число клеток в фигуре
  for j:=0 to POLE_HEIGHT-1 do   //- по всем строкам поля
    for i:=0 to POLE_WIDTH-1 do //- по длине строки
      if masPole[i][j]>0 then  begin //- живая клетка -> записать
координаты
        write (f,i); write (f,' '); writeln (f,j);
      end;
  //записать вид вселенной:
  if sbtFlat.down
  then i:=1
```

```
  else i:=0; write (f,i); write (f,' ');
  if sbtSphere.down
  then i:=1
  else i:=0;
  writeln (f,i);
  //записать соседей:
  if sbtSosedLU.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedU.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedRU.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedL.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedR.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedLD.down
  then i:=1
  else i:=0;
  write (f,i);
  write (f,' ');
  if sbtSosedD.down then i:=1
  else i:=0; write (f,i);
  write (f,' ');
  if sbtSosedRD.down
  then i:=1
  else i:=0;
  writeln (f,i);
  //записать правило рождения клеток - соседей, необходимых для
рождения:
  for i:=0 to 8 do begin
    if ((FindComponent ('sbtBirth'+inttostr(i)) as
TSpeedButton).down )
    then j:=1
    else j:=0;
    write (f,j);
    write (f,' ');
  end;
```

```
  writeln (f,'');
  //записать правило выживания клеток - соседей, позволяющих
продолжить жизнь
  for i:=0 to 8 do begin
    if ((FindComponent('sbtLife'+inttostr(i)) as
TSpeedButton).down)
    then j:=1
    else j:=0;
    write (f,j);
    write (f,' ');
  end;
  writeln (f,'');
  //записать макс. возраст - количество цветов, обозначающих
возраст:
  writeln (f,tbColor.position );
  //записать шлейф - количество цветов, обозначающих погибшие
поколения:
  writeln (f,tbShleif.position );
  //записать задержку:
  writeln (f,tbDelay.position );
  //записать комментарий к фигуре:
  s:=frmMemo.memo1.Lines.Text;
  if s<>'' then begin
    s:='//'+s;
    writeln (f,s)
  end;
  closefile(f);
  //showmessage(inttostr(i)+' '+inttostr(j));
  //вывести в заголовок название фигуры:
  caption:= NAME_PROG + '  [' + NameFig + ']';
  Refresh ;
  messagebeep(0)
end; //Save
```


По умолчанию данные записываются в *папку Figure* и имеют расширение *lif*:

```
Var
  //расширение файлов с фигурами:
  ext: string='lif';
```

Если текущая популяция ещё не сохранялась на диске и имя для неё не выбрано, то файл будет называться *temp*. Это обычный текстовый файл, поэтому конструировать и редактировать фигуры можно в любом текстовом редакторе. Например, для популяции (Рис. 6.9)

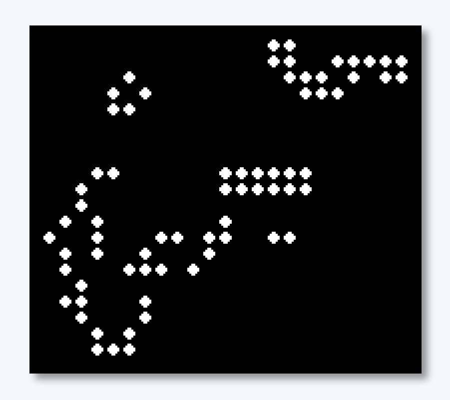


Рис. 6.9. Образцовая популяция

файл на диске будет таким:



```
70
40 16
. . .
31 35
1 0
1 1 1 1 1 1 1 1
0 0 0 1 0 0 0 0 0
0 0 1 1 0 0 0 0 0
1
0
10
//Тестовая популяция.
```

Сначала указывается общее число всех особей в популяции, а затем координаты всех клеток с живыми организмами. Сохраняется также информация обо всех настройках программы, о которых мы уже подробно говорили. Завершает файл комментарий из *Memo1*. Обратите внимание, что комментарий должен начинаться как и в *Delphi*, с двух слэшей.

Ничего нового и трудного в этой процедуре для вас нет, но вы должны иметь в виду, что на диске сохраняется текущая популяция из массива *masPole []*. Вероятно, не менее важно сохранить начальную популяцию и номер текущего поколения. Для этого следует завести ещё один массив того же типа, что и *masPole*:

```
type TPole = array[-1..POLE_WIDTH,-1..POLE_HEIGHT]of Integer;
```

В начале процедуры *TForm1.sbtGoClick* нужно скопировать массив с исходной популяцией в *новый* массив. Но здесь есть одна тонкость: в пошаговом режиме этот массив будет переписываться, так что, если вас интересует именно самая первая популяция, то заведите ещё и логическую переменную, которая будет сигнализировать о том, что массив с исходной популяцией уже сформирован.

Теперь разберём обратную процедуру – *загрузку* ранее сохранённой фигуры из файла. Для этого служит *кнопка sbtLoadFig* (**Hint**= Загрузить | Загрузить фигуру):

```
//КНОПКА Загрузить фигуру
procedure TForm1.sbtLoadFigClick(Sender: TObject);
var
  s, s2: string;
  found: integer;
  i:integer;
  SearchRec: TSearchRec;
begin
  if GameStatus='PLAY' then exit;
  //сформировать список файлов в папке FIGURE:
  s:=extractfilepath(application.exename)+'FIGURE\*.'+ ext;
  found:= FindFirst (s,faAnyFile,SearchRec);
  frmLoad.listbox1.items.Clear;
  while found=0 do begin
    s:=SearchRec.name;
    s2:='';
    i:=1;
    while (s[i]<>'.') and (i<=length(s)) do begin
      s2:=s2+s[i];
      inc(i)
    end;
    frmLoad.listbox1.items.add(s2);
    found:= FindNext (SearchRec)
  end;
  FindClose (SearchRec);
  nameFig:='';
  //показать форму со списком фигур:
  frmLoad.showmodal;
  if nameFig='' then exit; //фигура не выбрана

  //показать в заголовке название выбранной фигуры:
  form1.caption:= NAME_PROG + '   [' + NameFig + ']';
  //в информационной строке вывести число клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
  //скопировать комментарии к загруженной фигуре:
  frmMemo.memo1.Lines.Text:=frmLoad.memo1.Lines.Text;
  //поколение 0:
  nGen:=0;
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
  OutPole;
end;//sbtLoadFigClick
```


Очень просто загрузить фигуру на поле, но как выбрать нужную из множества сохранённых на диске? Постараемся облегчить игроку

проблему выбора. Для этого нам потребуется новая *форма frmLoad* (Рис. 6.10).

```
BorderStyle=bsSingle
Caption=  Выберите фигуру
Position=poMainFormCenter
```

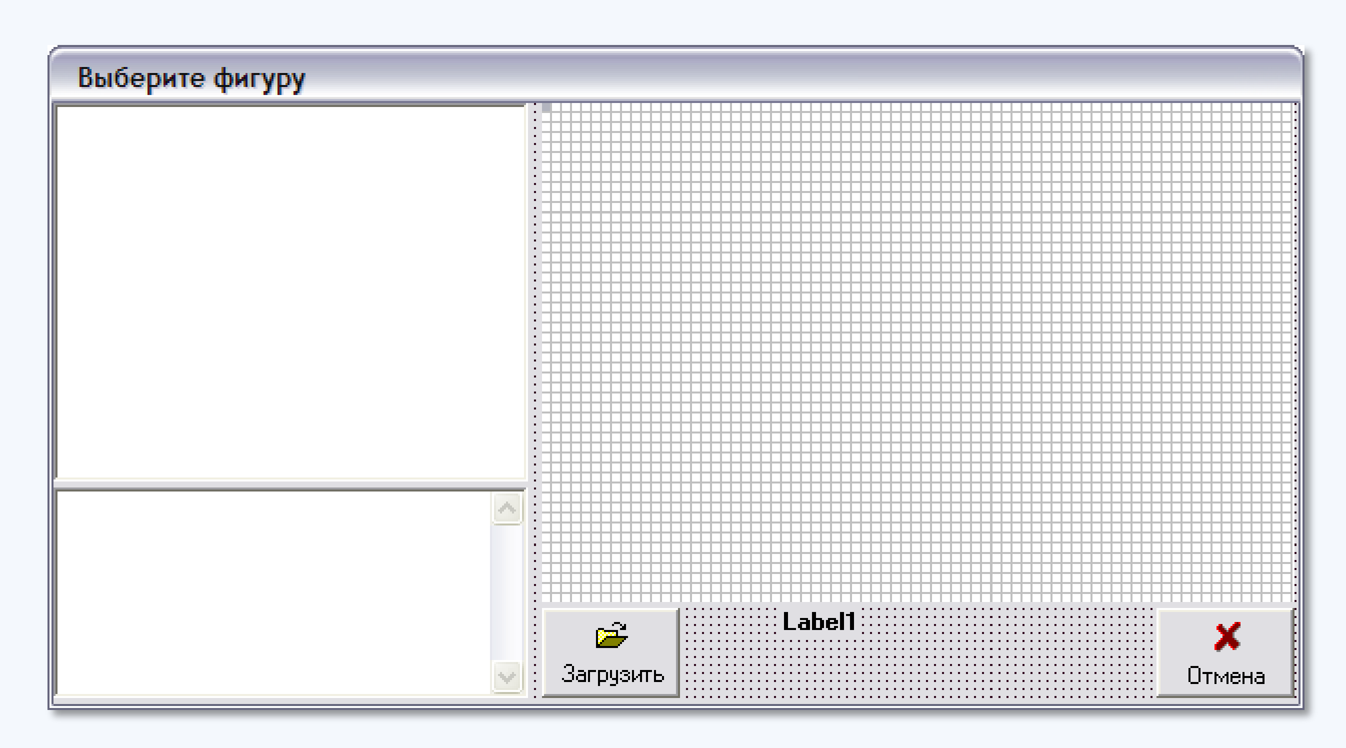



Рис. 6.10. Загрузочная форма

На ней мы расположим необходимые *элементы управления*.

В список *ListBox1* мы будем добавлять в алфавитном порядке (*Sorted=True*) названия всех файлов из *папки FIGURE*, имеющих заданное расширение.

В компоненте *Memo1* (*ScrollBars=ssVertical*) мы будем выводить *комментарий* к выбранной фигуре, подобно тому, как мы это делали в главном окне.

То же самое относится и к сетке *dgPole*, которая полностью аналогична сетке, отображающей игровое поле, только размер клеток у неё вдвое меньше:

```
DefaultColWidth=4
DefaultRowHeight=4
```

Метка *Label1* показывает число клеток в фигуре.
Кнопка *sbtLoad* (**Caption**=Загрузить) загружает выбранную фигуру.
Кнопка *sbtCancel* (**Caption**=Отмена) возвращает в вызывающую процедуру пустое имя файла – новая фигура не загружена!

```
//ОТМЕНИТЬ ЗАГРУЗКУ ФИГУРЫ
procedure TfrmLoad.sbtCancelClick(Sender: TObject);
begin
  nameFig:= '';
  close
end;
```

```
if nameFig='' then exit; //фигура не выбрана
```


После того как процедура *TForm1.sbtLoadFigClick* составит список всех файлов с фигурами:

```
frmLoad.listbox1.items.add(s2);
```

на экране появляется форма *frmLoad*:

```
  nameFig:='';
  //показать форму со списком фигур:
  frmLoad.showmodal;
  if nameFig='' then exit; //фигура не выбрана
```

В ней и нужно выбрать нужный файл из списка (Рис. 6.11).

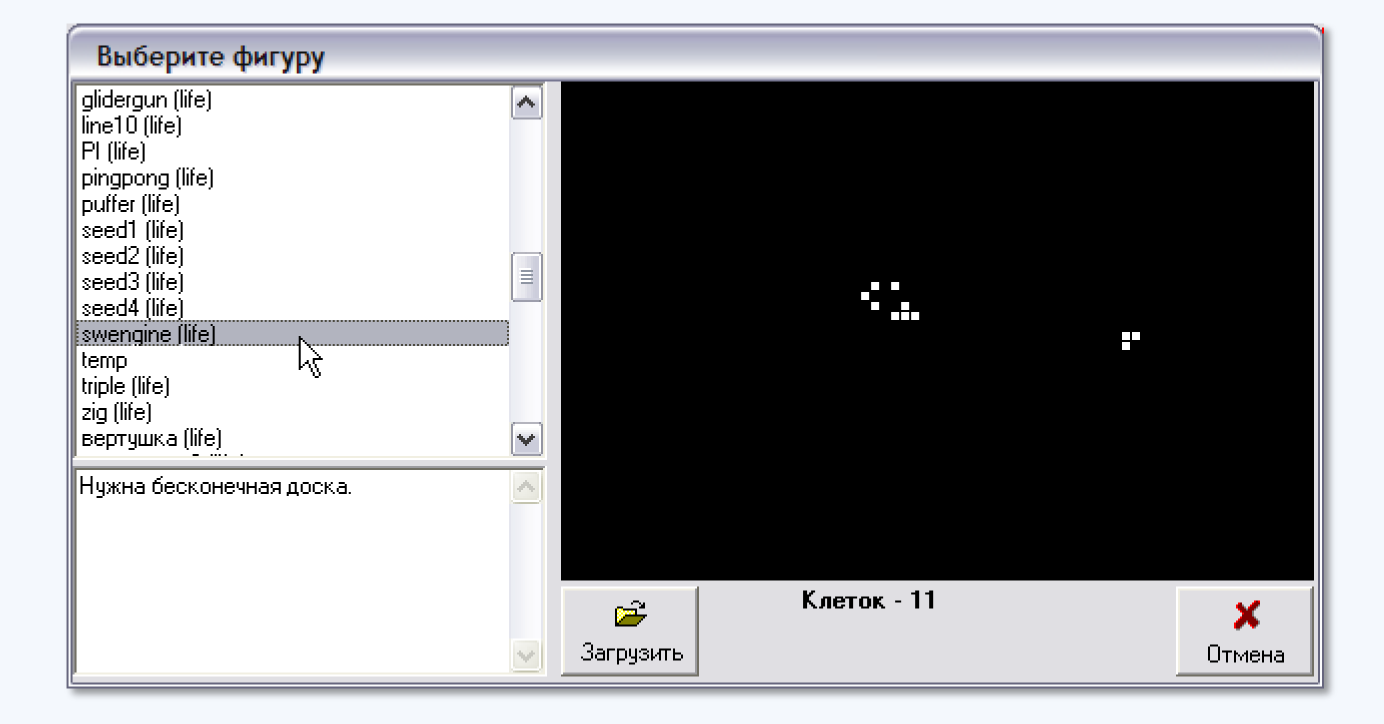


Рис. 6.11. Выбираем исходное поколение

Для этого, как обычно, нужно нажать *левую* кнопку мыши на названии файла:

```
//ВЫБРАТЬ ФИГУРУ ИЗ СПИСКА
```

```
procedure TfrmLoad.ListBox1MouseDown(Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  LifeUnit.nameFig:=
frmLoad.ListBox1.Items[listbox1.ItemIndex]+'.'+ ext;
  ViewFig;
end;
```

После чего новой фигуре присваивается имя выбранного файла, а сама фигура изображается на поле:


```
//ПОКАЗАТЬ ВЫБРАННУЮ ФИГУРУ
procedure ViewFig;
var
  s: string;
  F: TextFile;
  i: integer;
  x, y: integer;
  area: tRect;
begin
  s:=extractfilepath(application.exename)+'FIGURE\';
  NameFig:=s+nameFig; // - полный путь к фигуре
  {$i-}
  AssignFile(F,NameFig);
  Reset(F);
  {$i+}
  if IOResult<>0 then begin   {ошибка при загрузке файла}
    application.MessageBox ('Такой фигуры нет!',NAME_PROG, MB_OK);
    exit
  end;

  //очистить поле:
  ClearPole;
  //и комментарии:
  frmLoad.memo1.Clear;
  //считать количество клеток в фигуре:
  Readln(F, vsegoKl);
  //показать на экране:
  frmLoad.label1.Caption:= 'Клеток - ' + inttostr(vsegoKl);

  //считать фигуру:
  for i:=1 to vsegoKl do begin
    //считать координаты очередной клетки:
    Readln(F, x, y);
    //занести их в массив:
    masPole2[x,y]:=NUM_KLETKA;
    //и вывести на поле:
    frmLoad.dgPole.Canvas.Brush.Color:=COLOR_KLETKA;
    area:=frmLoad.dgPole.CellRect(x,y);
    frmLoad.dgPole.Canvas.FillRect(area);
  end;
```

```
  //считать вид вселенной:
   Readln(f,Flat,Sphere);
  //считать соседей:

Readln(f,SosedLU,SosedU,SosedRU,SosedL,SosedR,SosedLD,SosedD,Sosed
RD);
  //считать правило рождения клеток -
  //соседей, необходимых для рождения:
   for i:=0 to 8 do Read(f,Birth[i]);
  //считать правило выживания клеток -
  //соседей, позволяющих продолжить жизнь:
  for i:=0 to 8 do Read(f,Life[i]);
  //считать макс. возраст:
  Readln(f,Age);
  //считать шлейф - количество цветов,
  //обозначающих погибшие поколения:
  Readln(f,_shleif);
  //считать задержку:
  Readln(f,Delay);

  //считать комментарий к фигуре:
  s:='';
  while not eof(f) do begin
    Readln(F, S); //- считать строку из файла
    //комментарий к фигуре?
    if ((length(s)>1) and (s[1]='/') and (s[2]='/'))
    then begin //- комментарий
      s:=copy(s,3,length(s));
      frmLoad.memo1.Lines.Text:=s
    end
    else
     frmLoad.memo1.Lines.Text:=frmLoad.memo1.Lines.Text+#10+s
  end;
  //закрыть файл:
  closefile(f);
end;
```

Процедура очистки поля от предыдущей фигуры:

```
//ОЧИСТИТЬ ПОЛЕ
procedure ClearPole;
var
  i,j: integer;
begin
  frmLoad.dgPole.Canvas.Brush.Color:=COLOR_PUSTO;
  frmLoad.dgPole.Canvas.FillRect(frmLoad.dgPole.ClientRect );
  For i := 0 To POLE_WIDTH-1 do
    for j := 0 to  POLE_HEIGHT - 1 do
      masPole2[i,j]:=NUM_PUSTO;
  //клеток в фигуре:
  vsegoKl:=0;
```

```
  frmLoad.label1.Caption:= 'Клеток - 0';
end;
```

Мы подробно разобрали процесс сохранения файла, поэтому загрузка файла не должна вызвать у вас вопросов. Заметьте только, что клетки фигуры загружаются во вспомогательный массив *masPole2*:

```
var
  masPole2: TPole;
  vsegoKl: integer;
  Flat, Sphere: integer;//boolean;
  SosedLU,SosedU,SosedRU,SosedL,SosedR,SosedLD,SosedD,SosedRD:
integer;
  Birth: array[0..8] of integer;
  Life: array[0..8] of integer;
  Age: integer;
  Delay: integer;
  _shleif: integer;
```


Далее вы можете выбрать в списке любой другой файл, отменить загрузку новой фигуры или загрузить выбранную фигуру, если она вам понравилась:

```
//ЗАГРУЗИТЬ ФИГУРУ
procedure TfrmLoad.sbtLoadClick(Sender: TObject);
var i: integer;
begin
  //число клеток в фигуре:
  vsegoKl_inFig:=vsegoKl;
  //скопировать массив поля с выбранной фигурой:
  masPole:=masPole2;
  //установить вид вселенной:
  with Form1 do begin
    if Flat=1 then begin
       sbtFlat.down:= true;
       sbtSphere.down:= false
    end
    else begin
       sbtFlat.down:= false;
       sbtSphere.down:= true
    end;
    //установить соседей:
    if SosedLU=1
      then sbtSosedLU.down:= true else sbtSosedLU.down:= false;
    if SosedU=1
      then sbtSosedU.down:= true else sbtSosedU.down:= false;
    if SosedRU=1
      then sbtSosedRU.down:= true else sbtSosedRU.down:= false;
    if SosedL=1
      then sbtSosedL.down:= true else sbtSosedL.down:= false;
    if SosedR=1
```

```
      then sbtSosedR.down:= true else sbtSosedR.down:= false;
    if SosedLD=1
      then sbtSosedLD.down:= true else sbtSosedLD.down:= false;
    if SosedD=1
      then sbtSosedD.down:= true else sbtSosedD.down:= false;
    if SosedRD=1
      then sbtSosedRD.down:= true else sbtSosedRD.down:= false;
    //установить правило рождения клеток -
    //число соседей, необходимых для рождения:
    for i:=0 to 8 do
      if Birth[i]=1
        then (FindComponent ('sbtBirth'+inttostr(i)) as
TSpeedButton).down:= true
        else (FindComponent ('sbtBirth'+inttostr(i)) as
TSpeedButton).down:= false;
    //установить правило выживания клеток -
    //число соседей, позволяющих продолжить жизнь:
    for i:=0 to 8 do
      if Life[i]=1
        then (FindComponent ('sbtLife'+inttostr(i)) as
TSpeedButton).down:= true
        else (FindComponent ('sbtLife'+inttostr(i)) as
TSpeedButton).down:= false;
    //установить макс. возраст -
    //количество цветов, обозначающих возраст:
    tbColor.position:= Age;
    tbColorChange(Self);
    //установить шлейф - количество цветов,
    //обозначающих погибшие поколения:
    tbShleif.position:=_shleif;
    tbShleifChange(Self);
    //установить задержку:
    tbDelay.position:= Delay;
    tbDelayChange(Self);
  end;
  close
end;
```


В этой процедуре копируется игровое поле и значение переменных игры, а также устанавливаются в нужное положение элементы управления главного окна приложения в соответствии с их значениями в момент сохранения файла на диске.

Затем в заголовке окна выводится имя новой фигуры и другая информация:

```
  //показать в заголовке название выбранной фигуры:
  form1.caption:= NAME_PROG + '   [' + NameFig + ']';
  //в информационной строке вывести число клеток в фигуре:
  statusbar1.Panels[1].text:='Клеток: '+inttostr(vsegoKl_inFig);
  //скопировать комментарии к загруженной фигуре:
```

```
  frmMemo.memo1.Lines.Text:=frmLoad.memo1.Lines.Text;
  //поколение 0:
  nGen:=0;
  statusbar1.Panels[6].text:='Поколение: '+inttostr(nGen);
  OutPole;
```

Вы можете отредактировать фигуру, записать её на диск, изменить параметры игры и, наконец, запустить её эволюцию.

Есть ли жизнь на Марсе...
нет ли жизни на Марсе...
это науке неизвестно!

Комедия *Карнавальная ночь*

А теперь вернёмся в основную программу, чтобы нажать *кнопку sbtExit*



(**Hint**= Выход|Выход из программы),

```
//КНОПКА Выход из программы
procedure TForm1.sbtExitClick(Sender: TObject);
begin
  close
end;
```

чтобы закрыть её, а вместе с ней и последнюю страницу нашей книги о программировании игр и головоломок.



Исходный код программы находится в папке **LIFE**.

Факультатив 7. Бонусы, или Хитрая головоломка – хитори

Седьмого факультативного занятия не было в исходной рукописи, но *хитори* - это моя любимая интеллектуальная забава, о которой я даже написал книгу **[PB10]**. Надеюсь, она не разочарует и вас: хитори - одна из самых популярных японских головоломок в мире и одна из немногих, которую интересно решать и с карандашом в руке. По «решабельности» она значительно превосходит и пресловутое судоку, и какуро (*лично для меня*, как пел известный Перепелица).

Игровое поле в хитори – всегда квадратное и имеет размеры от 4 х 4 до 17 х 17 клеток (и даже больше). В каждой клетке записано число – от 1 до размерности поля (то есть при размерах поля 9 х 9 клеток это будут числа от 1 до 9, как в судоку). В некоторых строках и столбцах числа могут повторяться (Рис. 7.1). Таким образом, хитори – близкая родственница латинских квадратов, диаго и судоку.

2	2	5	1	1
2	1	4	2	3
3	4	3	2	5
3	5	1	5	5
4	5	2	3	1

Рис. 7.1. Нерешенная задача

2	2	5	1	1
2	1	4	2	3
3	4	3	2	5
3	5	1	5	5
4	5	2	3	1

Решенная задача

Цель решения хитори заключается в том, чтобы вычеркнуть *повторяющиеся* числа (или, иначе, закрасить клетку *черным* цветом) так, чтобы в каждой строке и каждом столбце все числа остались в одиночестве (то есть не повторялись). При этом в процессе решения задачи должны соблюдаться правила. А правил всего три:

Правило 1. Числа в *белых* клетках не должны повторяться ни в одной строке и ни в одном столбце. Если в *серых* клетках числа повторяются, то часть из них должна быть вычеркнута, то есть клетка должна стать *черной*.

Правило 2. Черные клетки не могут соприкасаться сторонами – только углами. На Рис. 7.1, справа это условие соблюдается.

*Правило 3. Белы*е клетки должны образовать *связную* область. Это значит, что из любой белой клетки должен существовать путь в любую другую белую клетку, проходящий через *стороны* белых клеток.

Чтобы различать клетки разного смысла, мы будем окрашивать их в разные цвета. *Серые* клетки – неразгаданные. На Рис. 7.1, слева мы видим задачу в исходном состоянии, когда ни одна клетка не разгадана.

Разгаданные клетки могут быть *белыми* и *черными*. В черных клетках находятся *вычеркнутые* числа, в белых – *невычеркнутые*, сохраненные. На Рис. 7.1, справа показана полностью решенная задача – все клетки белого и черного цвета.

Программированию хитори я отвел целую главу в книге ***Как решать** головоломные задачи на компьютере*. В ней я использовал язык *Си-шарп*, а нас сейчас куда больше интересует *Delphi*. Но поскольку изначально программа была написана на *Delphi* и именно она хорошо показала себя при подготовке книги, то я хочу предложить вам две ее разновидности. *Первая* - *rvHitori-Solver* – хорошо умеет *решат*ь задачи, но и только (Рис. 7.2).

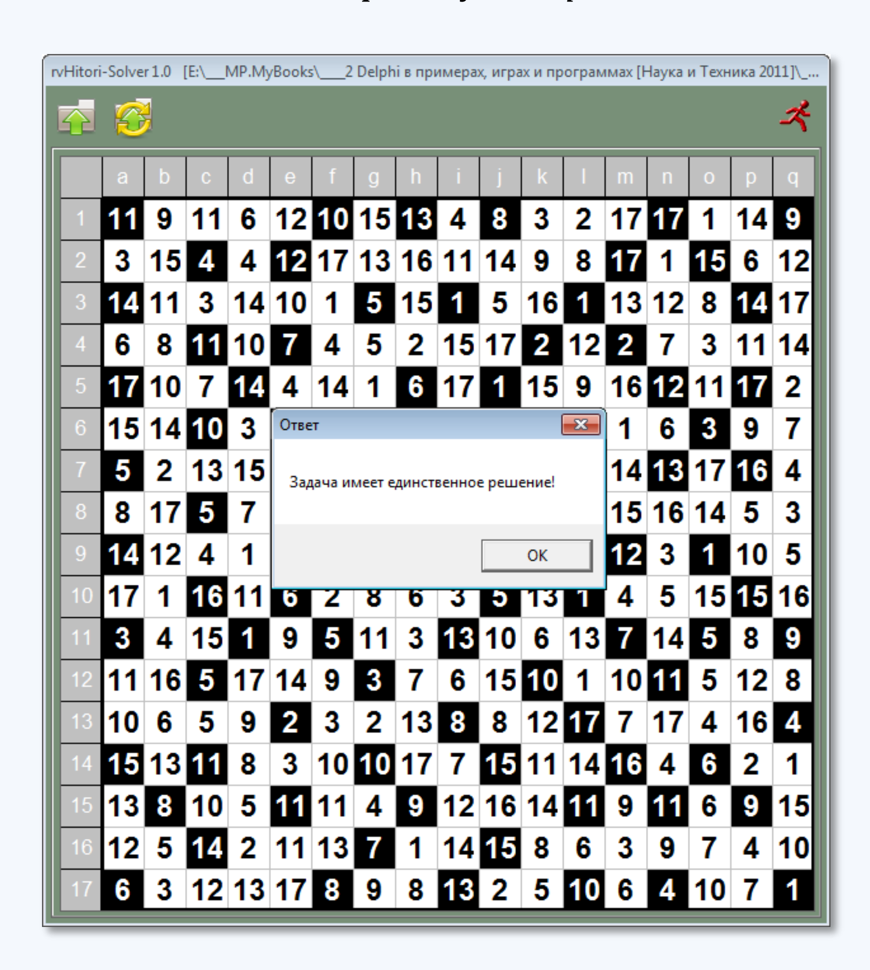


Рис. 7.2. *rvHitori-Solver* решает задачу

Вторая – *rvHitori* – не только решит предложенные ей задачи, но и поможет вам *составлять* свои собственные (Рис. 7.3)!

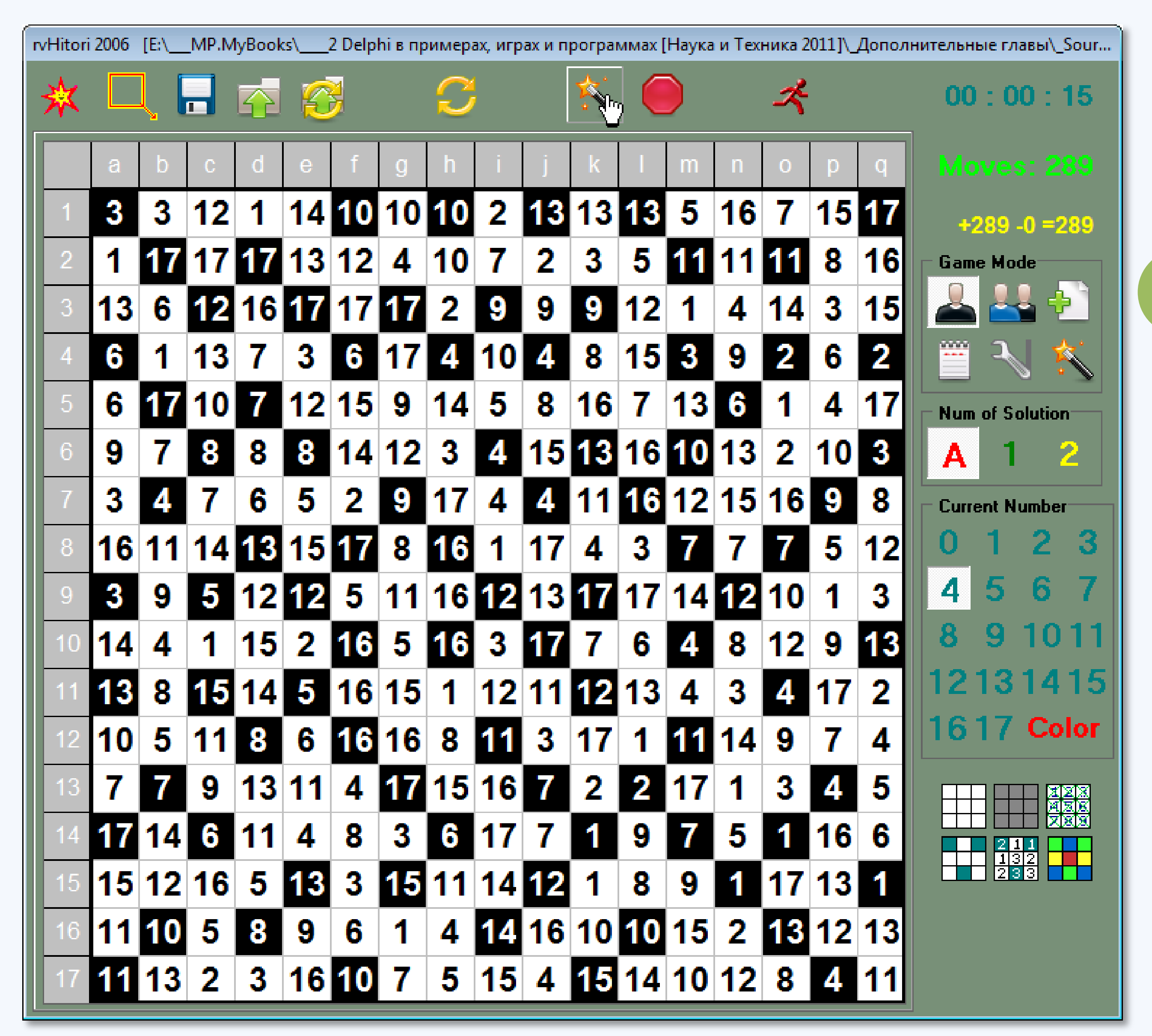


Рис. 7.3. *rvHitori* умеет всё!

Например, исключительно с помощью этой программы и были составлены все задачи для упомянутой выше добрым словом книги. А еще вы сможете решать задачи непосредственно в приложении, *не отходя от кассы*! А также составлять задачи *вручную*, пользоваться *подсказками* и прочими благами цивилизации.

В отличие от всех других «сольверов», программа *rvHitori* решает задачи не банальным перебором, а *логически*, то есть опираясь на статические и динамические приемы решения задач, которые ясно и подробно описаны в моей книге. На отдельной форме она ведет полный *протокол* решения за-

дачи и указывает приемы, которые были использованы для нахождения цвета той или иной клетки поля (Рис. 7.4).

```
Protocol
Move  272: l5  White [Dynamic4]
Move  273: b5  Black [Dynamic1.1]
Move  274: m6  Black [Dynamic1.1]
Move  275: k9  Black [Dynamic1.1]
Move  276: a14 Black [Dynamic1.1]
Move  277: c14 Black [Dynamic1.1]
Move  278: h14 Black [Dynamic1.1]
Move  279: k17 Black [Dynamic1.1]
Move  280: h4  Black [Dynamic1.2]
Move  281: q1  Black [Dynamic1.2]
Move  282: k16 White [Dynamic2]
Move  283: g5  White [Dynamic3]
Move  284: o6  White [Dynamic3]
Move  285: l16 Black [Dynamic1.1]
Move  286: g7  Black [Dynamic1.2]
Move  287: o4  Black [Dynamic1.2]
Move  288: m17 White [Dynamic3]
Move  289: f17 Black [Dynamic1.1]

All Solutions of the Puzzle are found - 1
Time  2.044 s

Save    Protocol...
Clear   ( ) Full
        (•) Right Moves Only
```

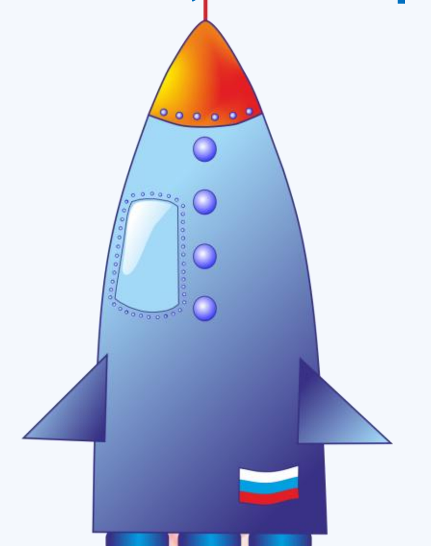
Рис. 7.4. Все ходызаписаны!

Вы легко можете представить себе размер кода, который описывает почти два десятка приемов решения задач хитори. Поэтому я предоставляю вам в полное распоряжение исходный код обеих программ – с надеждой и верой, что вы *самостоятельно* в нем разберетесь, потому что самостоятельность - самый лучший и правильный способ изучения программирования.



Исходный код программ находится в папках **_rvHitori-SolverRU** и **_rvHitori-SolverPro**.

END., или Карлсоны всегда возвращаются



- Шура, это конец!
- Это конец первой серии, студент!

Достойный ответ Остапа Бендера
меланхоликам и пессимистам

Сколько верёвочке ни виться,
а конец будет.

Уходя, гасите свет.

Житейское

Как известно, конец одного путешествия – это начало следующего! До новых приключений в стране *Delphi*!

Вперёд – труба зовёт!

Литература

[РВ10]		Рубанцев Валерий **Хитори**: круче, чем судоку Эксмо-Пресс, 2010. – 288 с. ISBN: 978-5-699-39575-0 Серия: Зарядка для ума
[РВ11]		Рубанцев Валерий **Delphi в примерах, играх и программах.** От простых приложений, решения задач и до программирования интеллектуальных игр Наука и Техника, 2011. – 672 с. ISBN: 978-5-94387-664-6 Серия: Самоучитель